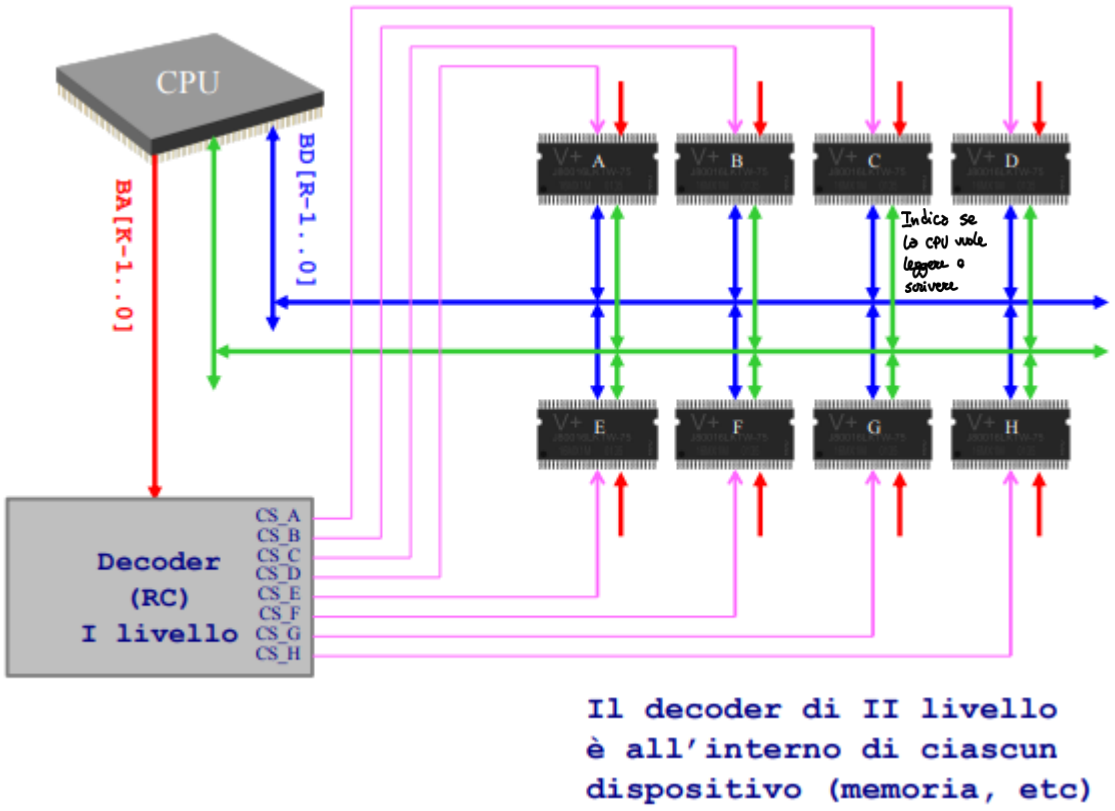
MAPPING E DECODIFICA

Una CPU emette un certo numero di indirizzi e altri segnali sui bus di sistema per comunicare con altri moduli.

Il numero di diversi indirizzi emessi dalla CPU costituisce lo **spazio di indirizzamento**. Una CPU che emette un indirizzo a 20 bit ha uno spazio di indirizzamento di 1 MB (2^20), mentre una CPU che emette un indirizzo a 32 bit ha uno spazio di indirizzamento di 4 GB (2^32). Oggi è consuetudine avere CPU con almeno 32 bit di indirizzo.



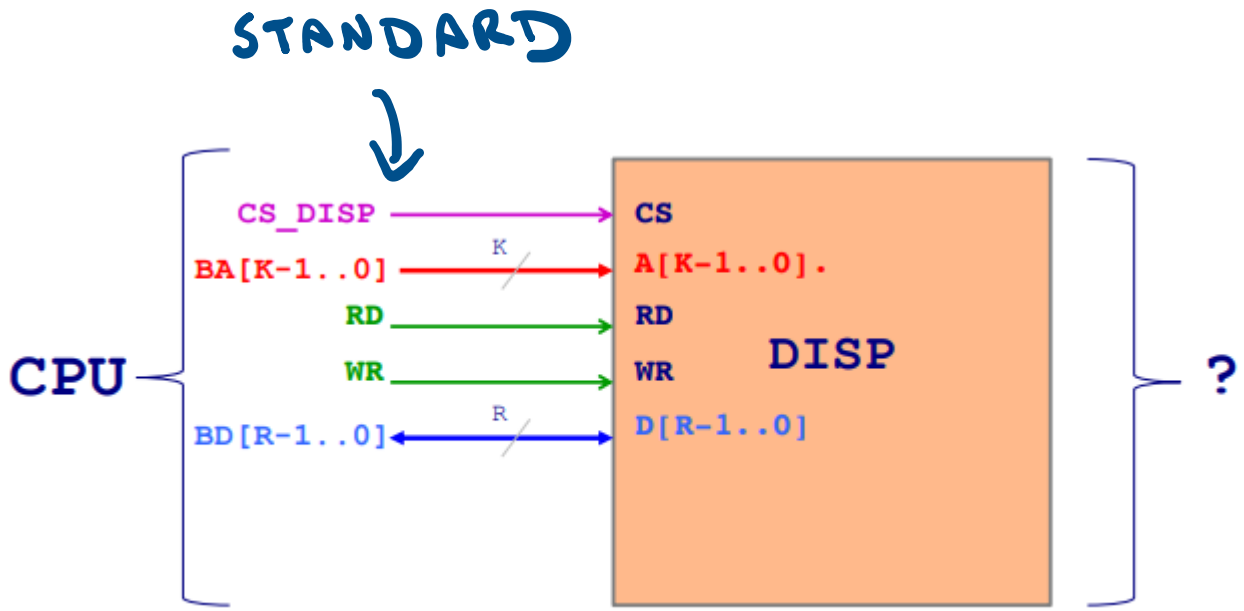
Vengono effettuate due decodifiche, una di primo e una di secondo livello.

Nella **decodifica di primo livello** il decoder seleziona il dispositivo attraverso l'indirizzo emesso dalla CPU. Nella **decodifica di secondo livello** il dispositivo capisce in che indirizzo la CPU vuole scrivere o leggere.

Condizione necessaria affinché un dispositivo fisico sia accessibile al software è che **il dispositivo deve essere mappato in uno spazio di indirizzamento.** Mappare in uno spazio di indirizzamento significa **associare al dispositivo una finestra di indirizzi di quello spazio di indirizzamento**. Si accede ai dispositivi mappati in uno spazio di indirizzamento con **cicli di bus.**

DISPOSITIVO GENERICO

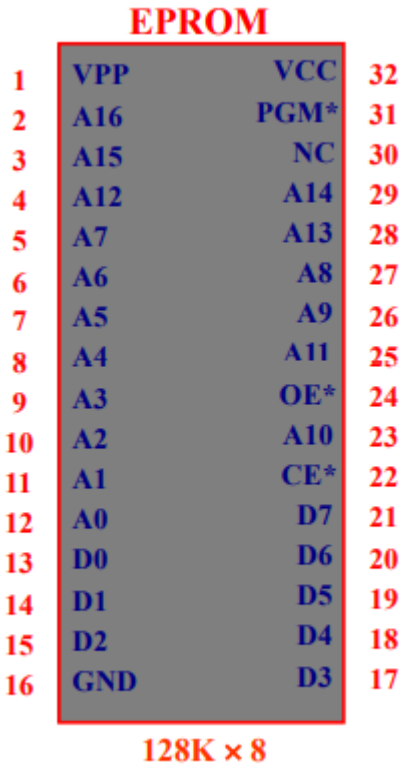
Un dispositivo qualsiasi comunica con la CPU mediante un'interfaccia standard. La comunicazione con l'esterno avviene secondo modalità specifiche del dispositivo, quindi non standard. I segnali BA[k-1…0] sono utilizzati per la decodifica di secondo livello.



MEMORIE EPROM

Memorie non volatili a sola lettura. La loro capacità è a multipli di 2 (32 k, 64 k, 128 k, …). È importante che esista la ROM per permettere l'avvio del sistema, che deve eseguire sempre lo stesso codice.

Le EPROM sono Erasable Programmable Read Only Memory, perciò possono essere scritte solo quando non sono in funzione.



VCC e GND – Servono per alimentare il dispositivo

PGM – Programming, indica quando la memoria viene riprogrammata

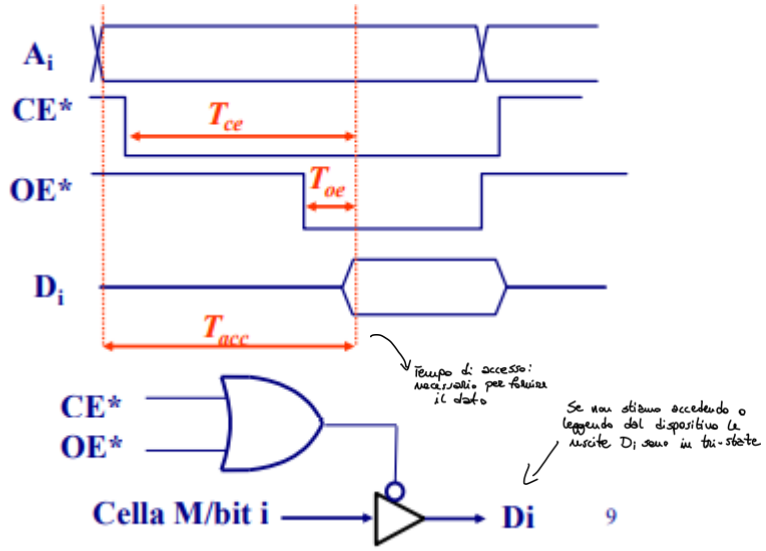
NC – Not connected

OE – Output enable (simile al READ negato)

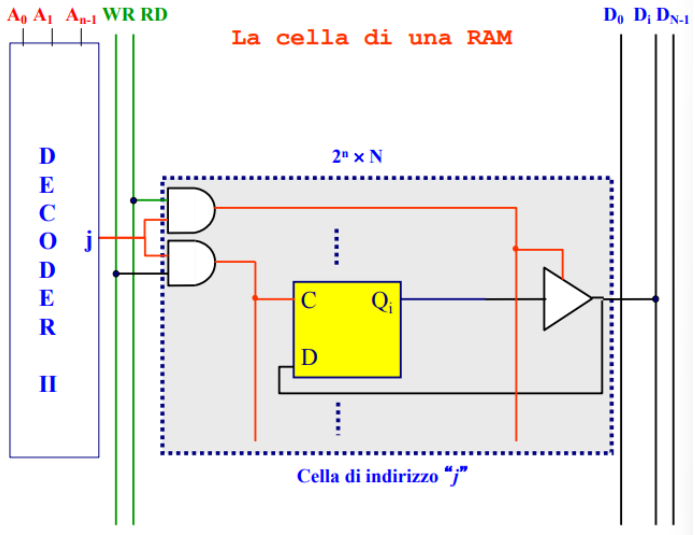
CE – Chip enable (Chip Select)

Di – Dati

Ai – Segnali di indirizzo



RAM



Nelle RAM ci sono entrambi i segnali di READ e WRITE. La decodifica di secondo livello viene fatta dal progettista del dispositivo. Sono memorie volatili, perciò i dati vengono persi se vengono disconnesse dall'alimentazione. Le DRAM (Ram Dinamiche) hanno 1 transistor per bit, maggiore capacità ma sono più lente.

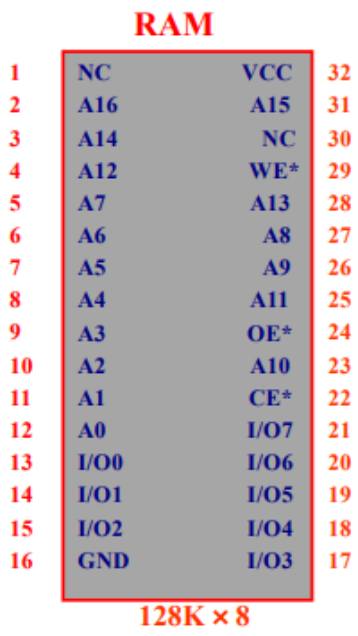
LETTURA

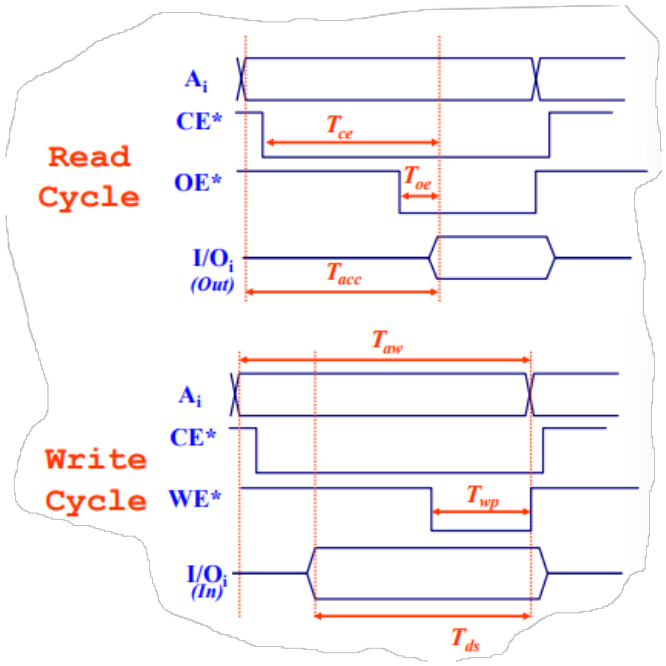
Se il segnale di READ è 1 viene attivato l'AND, perciò il dispositivo non è più in tri-state, ma è connesso alle uscite di dato. Il contenuto dei Latch CD viene connesso al bus di dati in uscita, pronto per essere letto.

SCRITTURA

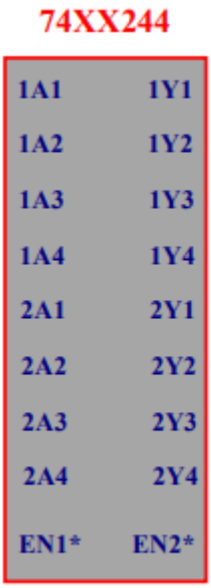
Il dispositivo è in tri-state. WRITE è 1 perciò viene scritto il contenuto dei Latch CD. Se READ è 0 Qi non è connesso elettricamente a Di.

Le RAM (Random Access Memory) saranno considerate sempre con un numero di indirizzi con potenza dispari (es. 2^5, 2^7 indirizzi, cioè 32k e 128k).

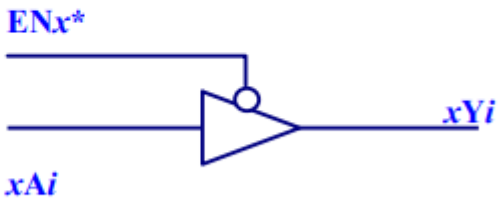




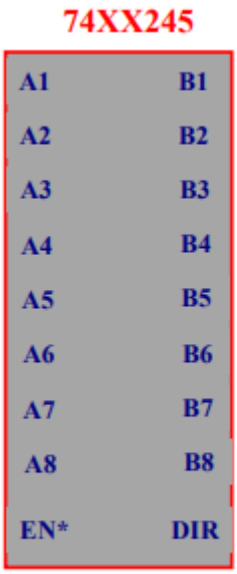
INTEGRATI NOTEVOLI – 244



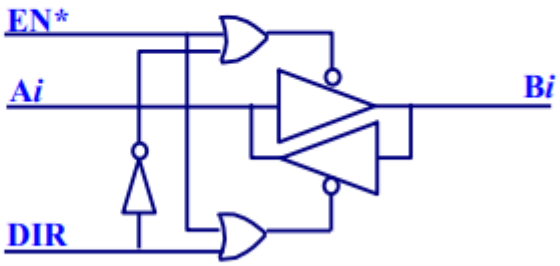
Driver 3-state ad 8-bit
(strutturato in 2 gruppi di 4 bit)



INTEGRATI NOTEVOLI – 245



Driver bidirezionale (*transceiver*) ad 8-bit.

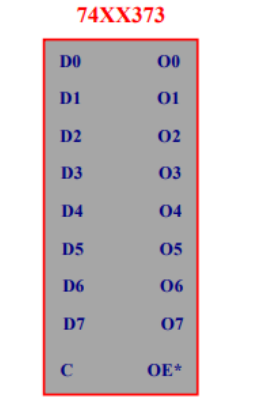


Il dispositivo 245 ha 8 transceiver. C'è un unico ENABLE per tutti e otto i transceiver. Lo stato del dispositivo è descritto dall'ENABLE.

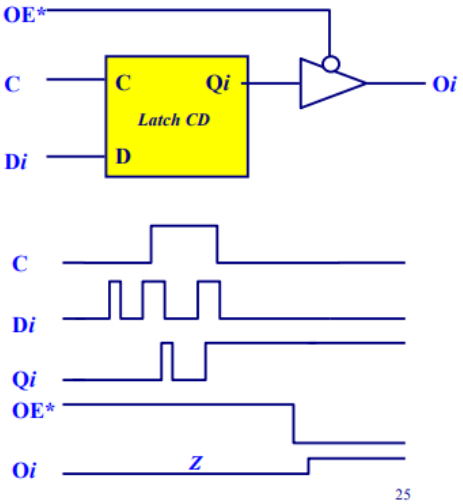
Ci sono 8 ingressi e 8 uscite.

Il dispositivo può scambiare informazioni sia verso destra che verso sinistra in base al valore del segnale DIR.

INTEGRATI NOTEVOLI – 373



Latch a 8-bit con uscite 3-state

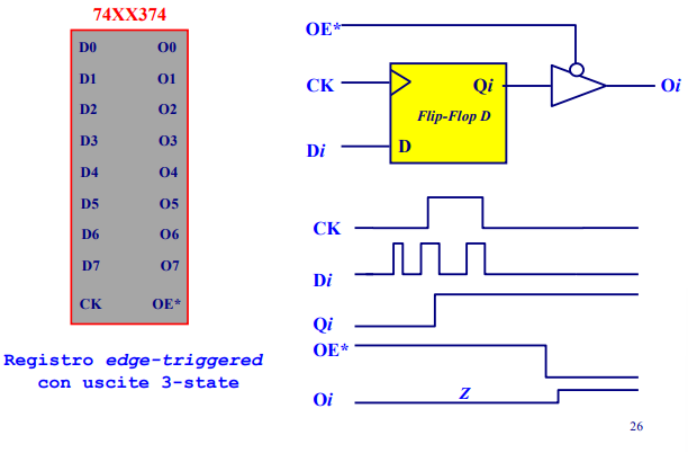


Il dispositivo è formato da otto Latch-CD. C'è un segnale di controllo C per ogni Latch-CD. Il segnale D è quello attraverso cui entra l'informazione. Uno per ogni Latch-CD.

Quando C=1 il Latch-CD acquisisce l'informazione presente sull'ingresso D. Quando C=0 il Latch-CD non acquisisce l'informazione, ma mantiene il valore precedentemente acquisito.

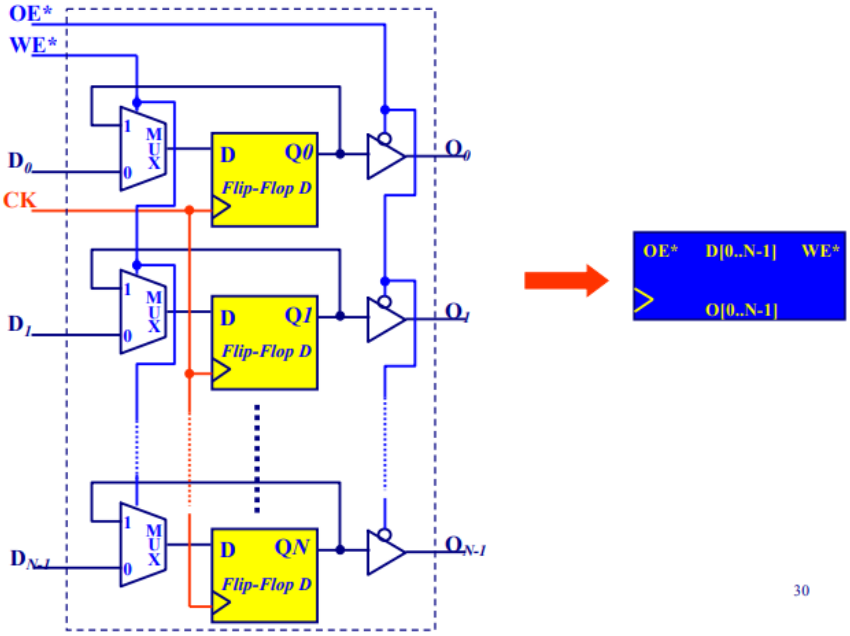
Tramite l'**output enable** il dispositivo può essere messo in tri-state. C'è un solo **output enable** per tutti e otto i possibili dispositivi in 3-state, ognuno associato a un latch-CD.

INTEGRATI NOTEVOLI – 374



È un dispositivo integrato con al suo interno 8 componenti, in particolare 8 FF-D.
I flip-flop D campionano il segnale D durante un fronte di salita del clock CK. I segnali di ingresso sono 8, D[7…0], mentre quelli di uscita, sempre 8, Q[7…0].
Le uscite dei FF-D sono collegate tramite gli 8 3-state alle uscite O[7…0]. Le uscite dei FF-D acquisiscono il valore dell'ingresso D con un ritardo espresso sul datasheet. È presente un unico segnale di clock CK per tutti i FF-D e idem per quanto riguarda il segnale di **output enable.**
Quando OE'=1 l'uscita Q[7...0] non è elettricamente connessa all'uscita O[7…0]. In questa situazione il dispositivo è in 3-state.
A volte può essere utile avere la possibilità di mettere il dispositivo in 3-state.
Quando OE'=0 l'uscita Qi è elettricamente connessa all'uscita (o PIN) Oi.

REGISTRO EDGE-TRIGGERED CON WE*



È utile realizzare registri che siano **edge triggered**, ossia che campionino informazioni in ingresso solo ai fronti di un segnale di clock CK. In particolare, assumeremo che il campionamento avvenga durante i fronti di salita del segnale di clock.

Abbiamo bisogno di un registro a N bit che è in grado di:

- Acquisire N dati sugli ingressi D[N-1…0] sul fronte di salita del segnale di clock CK
- Abilitare le uscite degli N FF-D (ossia mentre il dispositivo è in 3-state o meno)

Nel nostro esempio:

- Se WE'=0 allora significa che voglio scrivere ciò che è presente sugli ingressi al prossimo fronte di salita del segnale di clock CK.
- Se WE'=1 allora significa che non voglio scrivere ciò che è presente sugli ingressi. Dunque, negli N FF-D verrà mantenuto il precedente valore/informazione.

Poniamo WE' come bit di indirizzo di un MUX. Anche WE' è un ingresso.

Il segnale D deve essere stabile per un periodo prima del fronte di salita del clock e anche dopo il fronte di salita del clock (**set-up time e hold time**).

In un registro, oltre a poter decidere quando per scegliere (con WE'), è utile poter mettere le uscite in 3-state.

L'**output enable (OE)** è un unico segnale che agisce sugli N 3-state.

Quando OE'=0 le uscite Qi sono connesse elettricamente agli Oi.

Quando OE'=1 le uscite Qi non sono connesse elettricamente agli Oi (**3-state**).

REGISTER FILE (1 WRITE – PORT, 1 READ – PORT)

MAPPING DI DISPOSITIVI DA 8 BIT IN SISTEMI CON BUS DATI DA 8 BIT

Consideriamo dispositivi con porta dati a 8 bit.

Imponiamo temporaneamente l'ulteriore condizione che il parallelismo del bus dati sia a 8 bit.

In queste ipotesi l'assegnamento a un dispositivo di una finestra di indirizzi in uno spazio di indirizzamento avverrà in generale nel rispetto delle due seguenti condizioni restrittive:

- La dimensione della finestra di indirizzi associata a un dispositivo è una potenza di due
- La finestra è composta da indirizzi contigui

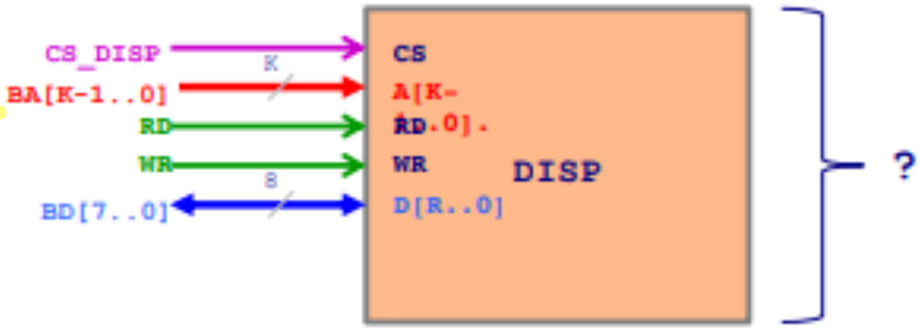
Un dispositivo accessibile attraverso il bus occupa in generale n= 2^k posizioni nello spazio di indirizzamento. N rappresenta il numero di oggetti a 8 bit indirizzabili all'interno del dispositivo (es. numero di celle di memoria nelle RAM ed EPROM).

K è il numero di bit di indirizzo interni al dispositivo. È fortemente variabile al variare del dispositivo: in generale nei dispositivi di I/O K è piccolo, mentre nei dispositivi di memoria K è grande.

CARATTERISTICHE AI MORSETTI DI UN DISPOSITIVO INDIRIZZABILE SU UNA FINESTRA DI 2^K BYTE

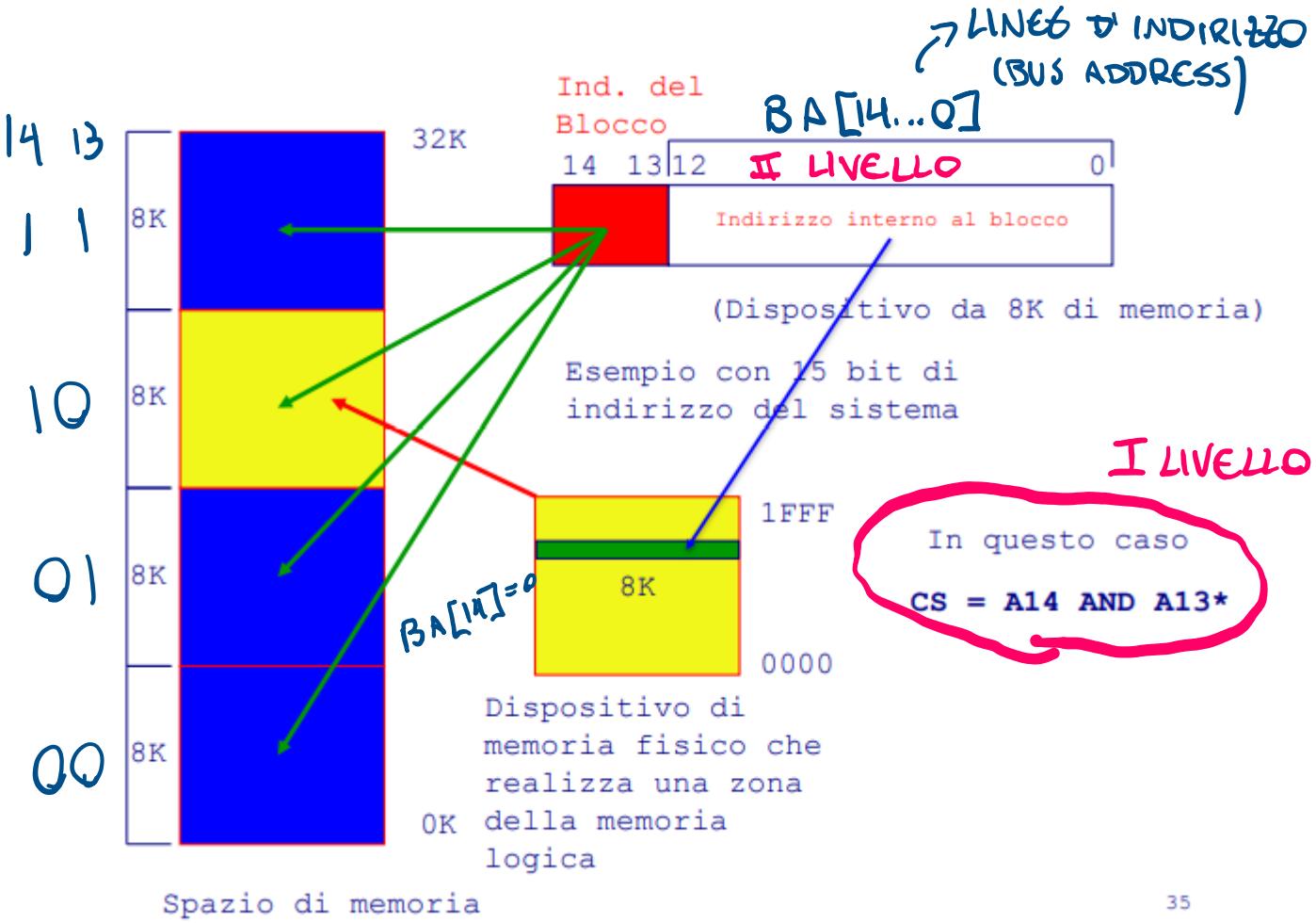
Qualunque dispositivo da 8 bit con all'interno n=2^k elementi indirizzabili separatamente ha al suo interno un decoder (II livello) di K variabili con ingresso di enable che seleziona i singoli oggetti indirizzabili.

- **READ**, detto anche **OE** è il comando di lettura. Quando READ e CS sono attivi, il dispositivo espone su BD[7…0] il contenuto della cella indirizzata.
- **WRITE** è il comando di scrittura. Quando CS asserito sul fronte di discesa di WR è campionato il dato presente su BD[7…0]



Il controllo di tutto è dato dalla CPU e chi scrive il codice che la CPU deve eseguire.

Se la CPU può fare dei calcoli al suo interno il tutto è più veloce perché non intervengono i cicli di bus.



Se vogliamo mappare un dispositivo da 8k in questo spazio di indirizzamento è buona norma mappare il dispositivo a indirizzi multipli della taglia del dispositivo.

Una volta individuato il dispositivo con la decodifica di primo livello entra in gioco la decodifica di secondo livello, che viene fatta all'interno del dispositivo. Ci servono 13 bit per individuare uno spazio di indirizzamento all'interno del dispositivo, che sono proprio i 13 bit che avanzano da BA[14…0].

Sembra che queste azioni vengano svolte in modo sequenziale, ma in realtà il processore mette a disposizione tutti e 15 i bit di indirizzo.

A basso livello, spesso, se si può fare le cose insieme per velocizzare il tutto, lo si fa.

Se si considera un dispositivo D di n=2^k byte indirizzabili, si dice che **D è mappato all'indirizzo A** se gli indirizzi dei byte di D sono compresi tra A e A+(n-1), cioè se A è l'indirizzo più basso tra tutti gli indirizzi associati a D.

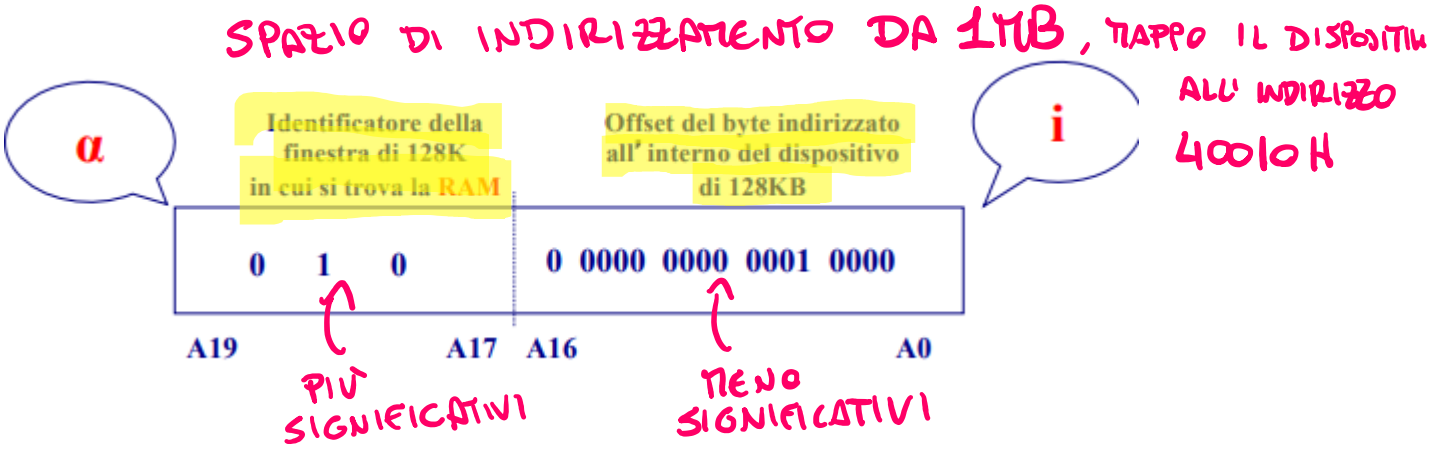
Si dice che **D è allineato** se A è un multiplo di n (numero di byte interni al dispositivo), cioè se (indirizzo più basso di D) % n = 0

Se D è allineato allora i bit meno significativi di A sono uguali a 0.

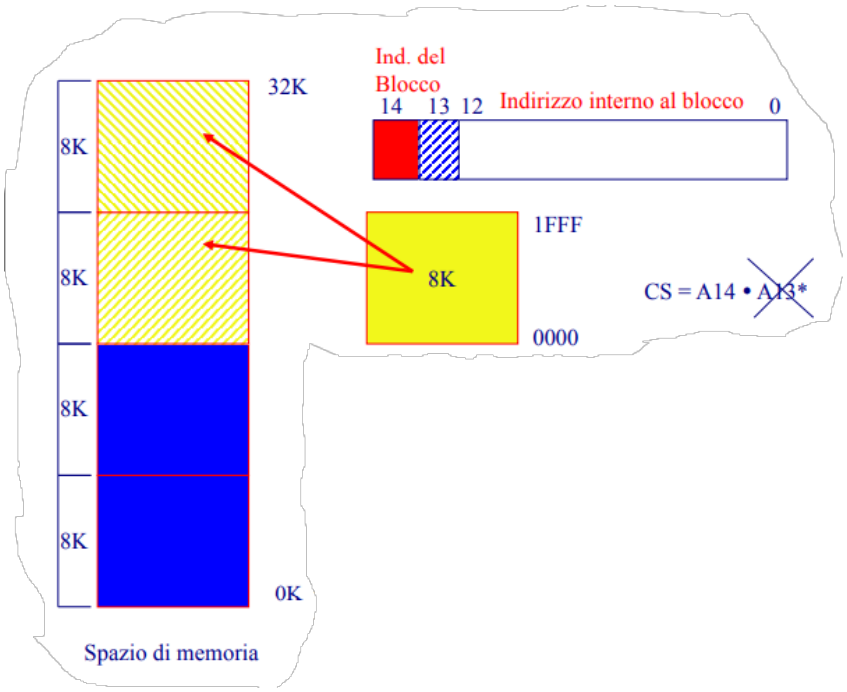
Es. un dispositivo a due byte è allineato se è mappato a un indirizzo pari, un dispositivo a 8 byte è allineato se è mappato a un indirizzo il cui valore codificato in binario termina con 3 zeri. Un dispositivo a 64 kb è allineato se il suo indirizzo in codice esadecimale ha le quattro cifre meno significative uguali a 0.

È inutile mappare un dispositivo in modo non allineato.

Per individuare univocamente una finestra allineata di 2^k byte in uno spazio di indirizzamento, suddividiamo l'indirizzo in una parte più significativa e una meno significativa. La parte più significativa individua le finestre allineate di 2^k byte presenti nello spazio di indirizzamento, mentre la parte meno significativa individua l'offset nel chip del byte indirizzato.



DECODIFICA SEMPLIFICATA



Il modulo di memoria fisico è di fatto attivato in due differenti zone della memoria logica.

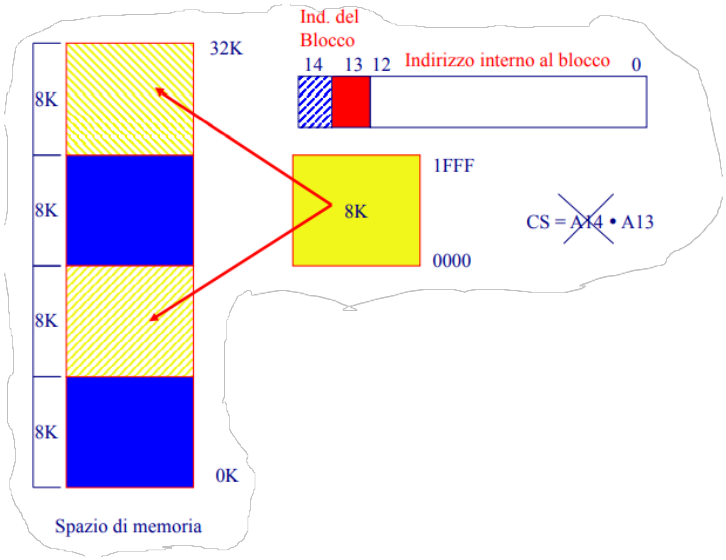
Se gli indirizzi usati da un programma sono quelli che vanno da 16K a 24K e quelli da 24K a 32K non sono usati, allora è possibile la decodifica incompleta o parziale in quanto la zona 24-32 non viene mai indirizzata. L'espressione del **chip select** è più semplice.

Questa pratica offre un piccolo risparmio quando ci sono molte linee di indirizzo. Con la decodifica semplificata si crea una sorta di alias ai dispositivi e si semplifica un po' la decodifica di primo livello.

Non è possibile, in un sistema del genere, che allo stesso indirizzo si attivino più chip select.

DECODIFICA PARZIALE 2/2

Il modulo di memoria fisico è attivato in due differenti zone della memoria **non consecutive**.

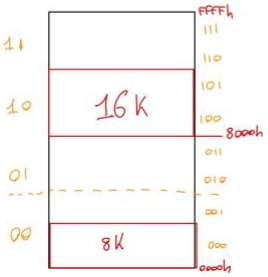


Esercizio

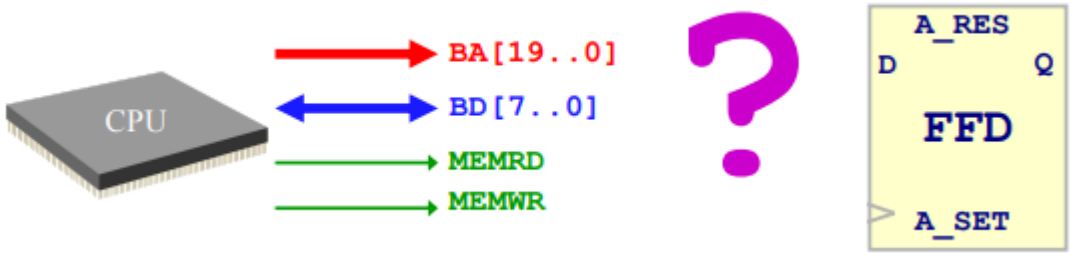
Si consideri un sistema con bus indirizzi a 16 bit e bus dati a 8 bit. Scrivere le espressioni di decodifica completa e semplificata (quella da usare all'esame) nei seguenti casi:

1) **Dispositivo di memoria da 16 KB mappato a 8000h**
2) **Dispositivo di memoria da 8 KB mappato a 0000h**
3) **Entrambi i dispositivi precedenti**

Se c'è un solo dispositivo (casi 1 e 2) il CS è molto particolare….



MAPPING, READ, WRITE E SET/RESET DI UN FF-D



Il testo dice di mettere la RAM e la EPROM a determinati indirizzi. Dobbiamo spezzare la RAM da 64k in due RAM da 32k.

Se dobbiamo leggere dal FF-D dobbiamo collegare in qualche modo il segnale di uscita Q del FF-D al segnale di bus dati BD[7…0]. Dovremo sicuramente usare un 3-state.

Per quanto riguarda invece la scrittura, essa dovrà passare sempre dal bus dati, che dovrà essere collegata all'ingresso D del FF-D. Dobbiamo fare in modo che il dato sia stabile per il su-time e l'h-time. Noi sappiamo che in un ciclo di bus ci sono più cicli di clock, perciò dovremo usare il segnale MEMWR.

Per impostare A_RES e A_SET a 1 (non contemporaneamente) dobbiamo fare in modo che ciò avvenga durante il ciclo di bus.

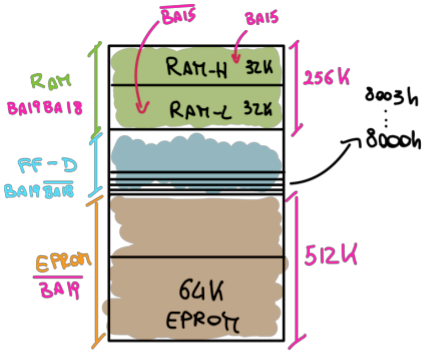
Assumiamo che i comandi del FF-D siano mappati ai seguenti indirizzi:

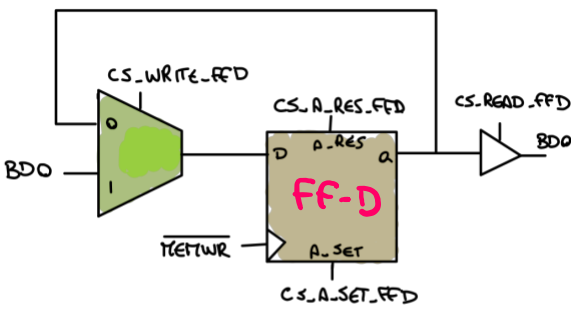
CS_READ_FFD -> 80003h

CS_WRITE_FFD -> 80002h

CS_A_RES_FFD -> 80001h

CS_A_SET_FFD -> 80000h





$CS\_READ\_FFD = BA19\ \overline{BA18}\ BA1\ BA0 \cdot MEMRD$

$CS\_WRITE\_FFD = BA19\ \overline{BA18}\ BA1\ \overline{BA0}$

$CS\_A\_RES\_FFD = BA19\ \overline{BA18}\ \overline{BA1}\ BA0 \cdot MEMWR$

$CS\_A\_SET\_FFD = BA19\ \overline{BA18}\ \overline{BA1}\ \overline{BA0} \cdot MEMWR$

Voglio che il segnale vada a 1 solo durante il ciclo di bus di lettura

Dovremo usare il singolo segnale BD0 del bus dati per leggere e scrivere il singolo bit di dato.

Connettiamo l'uscita del FF-D al bus dati attraverso un 3-state con segnale CS_READ_FFD.

Vado ad aggiornare il contenuto del FFD solo al fronte di discesa di MEMWR perché così do il tempo al bus dati di essere pronto (aumentiamo il tempo di setup).

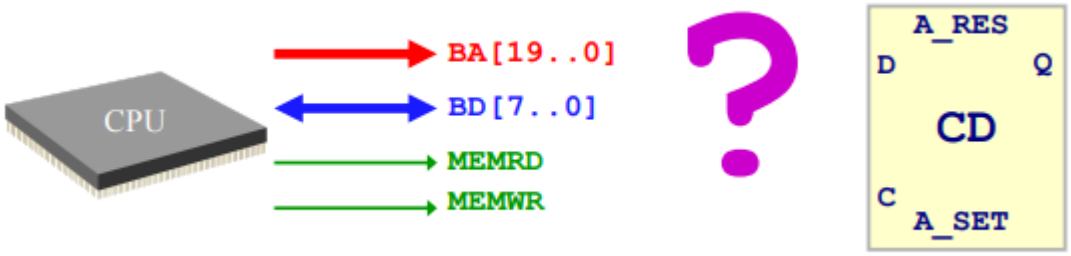
I segnali asincroni vanno utilizzati con molta cautela, quasi mai. Vengono utilizzati principalmente all'avvio del sistema per far sì che tutta la logica venga impostata secondo il nostro volere.

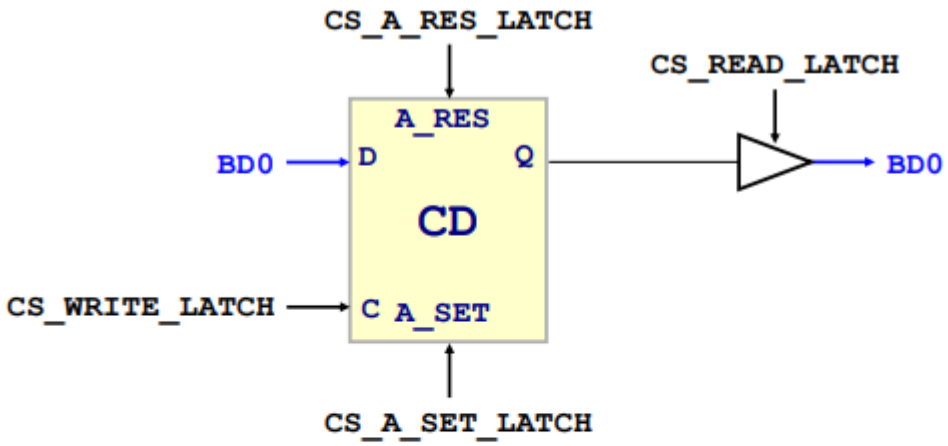
I segnali asincroni sono condizionati con il MEMWR perché se non lo fossero c'è il rischio che, se anche per un periodo brevissimo il segnale andasse a 1 in transitorio, avrebbe effetti indesiderati sul FF-D. Perciò i segnali asincroni avranno effetto solo se durante un ciclo di scrittura vado a scrivere su quegli indirizzi.

MAPPING, READ, WRITE E SET/RESET DI UN LATCH

Con una CPU possiamo scrivere, leggere nel latch e settare o resettare in modo asincrono il latch.

Consideriamo una CPU con bus dati a 8 bit con 20 bit di indirizzo. 64k di RAM agli indirizzi alti e 64K di EPROM agli indirizzi bassi.






```
CS_RAM_H        = BA19·BA18·BA15
CS_RAM_L        = BA19·BA18·BA15*
CS_READ_LATCH   = BA19·BA18*·BA1·BA0·MEMRD  (ist. lettura)
CS_WRITE_LATCH  = BA19·BA18*·BA1·BA0*·MEMWR (ist. scrittura)
CS_A_RES_LATCH  = BA19·BA18*·BA1*·BA0·MEMWR (ist. scrittura)
CS_A_SET_LATCH  = BA19·BA18*·BA1*·BA0*·MEMWR (ist. scrittura)
CS_EPROM        = BA19*
```

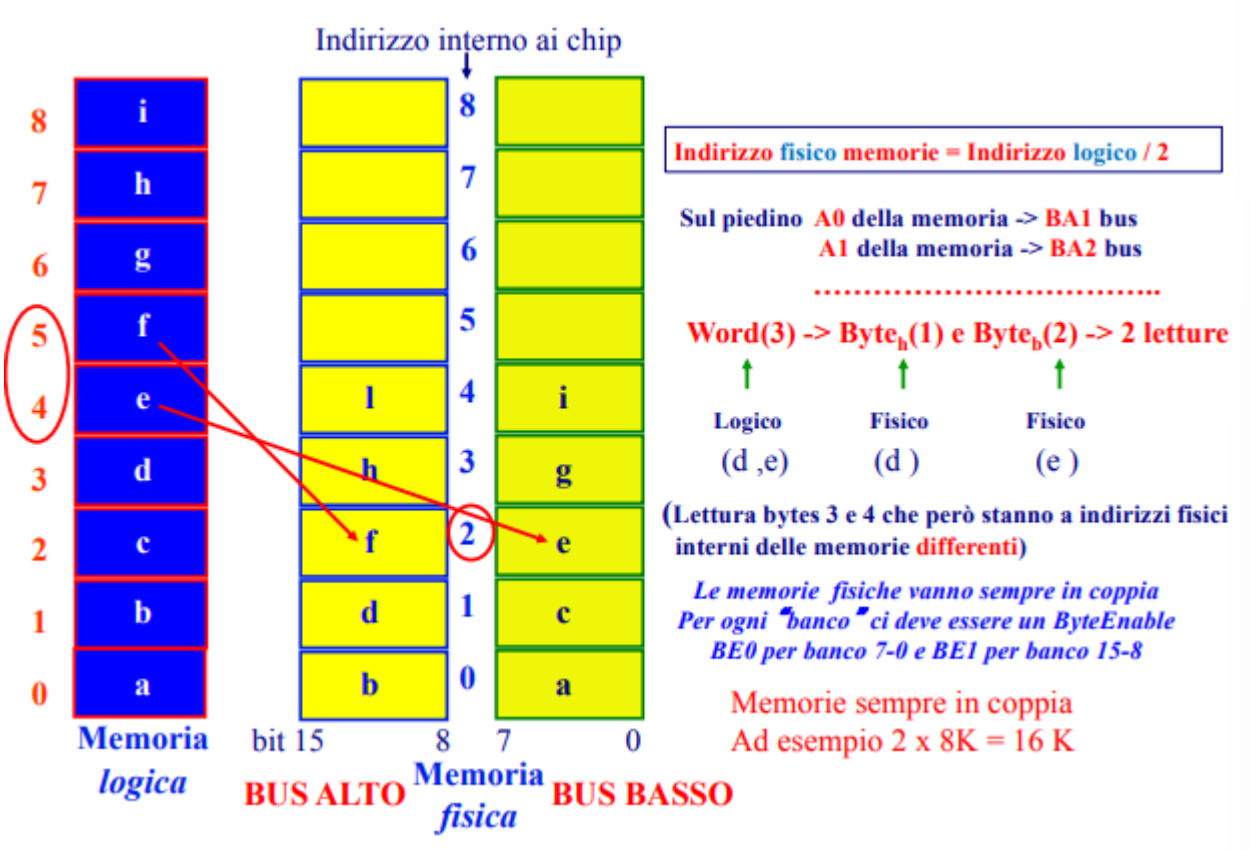
INCREMENTARE IL PARALLELISMO DEI DATI

Abbiamo considerato fino a ora sistemi con un parallelismo (bus dati) a 8 bit. Ogni trasferimento richiede un ciclo di bus (N byte -> N cicli di bus).

Per rendere il tutto più efficiente possiamo aumentare il parallelismo dei dati, riducendo la dimensione di ciascuna memoria e trasferendo più dati nello stesso ciclo di bus. All'aumentare dei fili del bus dati si riduce la taglia dei dispositivi connessi.

La cosa da non fare è trasferire gli elementi sequenzialmente in memorie più piccole. Gli elementi contigui vanno su memorie diverse.

MEMORIA CON PROCESSORI A PARALLELISMO A 16 BIT



Le memorie hanno sempre 8 bit di dato. Quella blu è la memoria logica, cioè un insieme di celle in cui a ciascun indirizzo c'è un byte. Dobbiamo riorganizzare la memoria logica per distribuirla sui due bus dati (gialli). Il metodo giusto è quello di utilizzare non più una memoria, ma due memorie fisiche, una collegata al bus dati basso e una al bus dati basso.

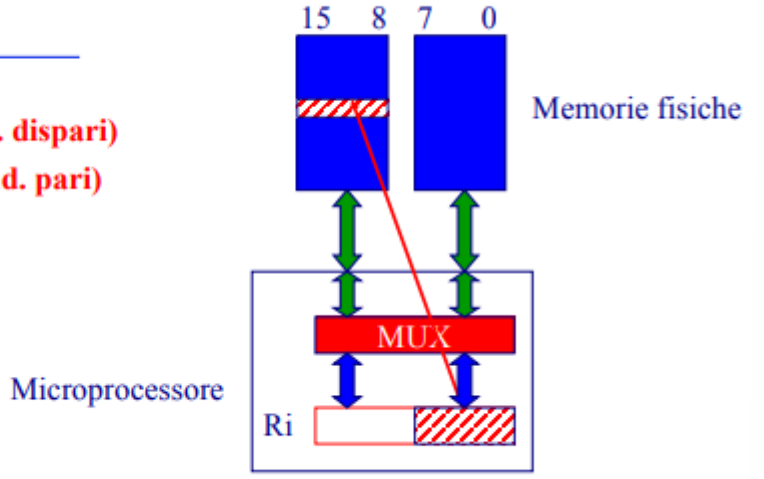
Se voglio trasferire 16 bit, durante il trasferimento entrambe le memorie devono essere attive.

Gli indirizzi di A e B sono all'indirizzo, rispettivamente, 0 e 1. Eliminando l'ultimo bit sono entrambi all'indirizzo 0.

Le memorie fisiche vanno sempre in coppia. Per ogni "banco" ci deve essere un ByteEnable BE0 per banco 7-0 e uno BE1 per banco 15-8. Anche le memorie sono sempre in coppia. (es. 2 x 8k = 16k).

Per leggere due dati che si trovano a due indirizzi fisici diversi devo fare due letture. I processori RISC non riescono a fare questo tipo di operazioni; perciò, è necessario memorizzare i dati a indirizzi pari.

BE1	BE0	
1	1	Word
1	0	Byte alto (ind. dispari)
0	1	Byte basso (ind. pari)
0	0	Non possibile



Se BE1 e BE0 sono entrambi 1 il trasferimento è a 16 bit, se solo uno dei due è 1 il trasferimento è a 1 byte.

Il MUX smista tra bus dati e registri interni. Ciò avviene sia in lettura che in scrittura.

Es. leggo un singolo byte sul bus dati alto, che finirebbe nella parte alta, ma dato che sto facendo una lettura a 8 bit, è più consono che il byte finisca nella parte bassa.

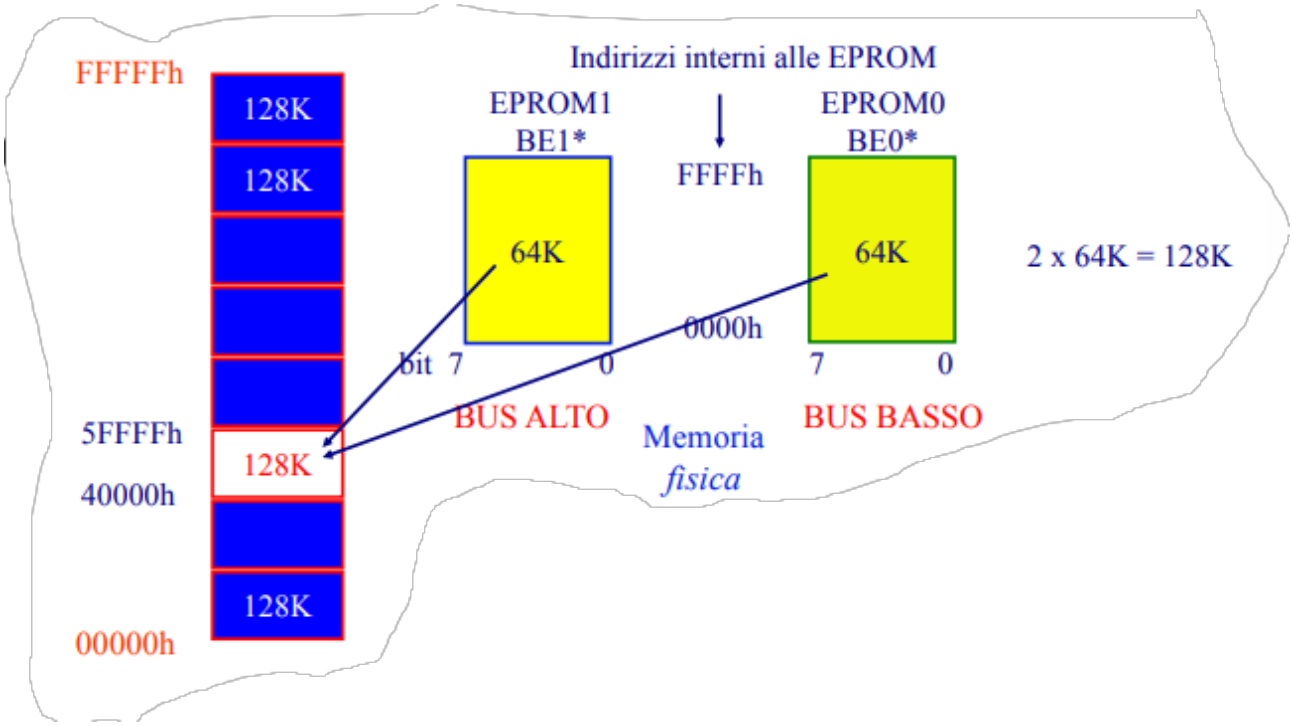
BE1 e BE0 ci dicono se è coinvolto il bus dati 7…0 o quello 15…8.

È possibile, a partire dalle 16 linee di indirizzo, generare BE0 e BE1?

Es. ho un indirizzo 1001111111110101, posso generare BE0 e BE1?

Se dovessi trasferire sempre e solo un byte, l'ultimo bit BA0 mi dice se devo andare sul bus dati alto o su quello basso.

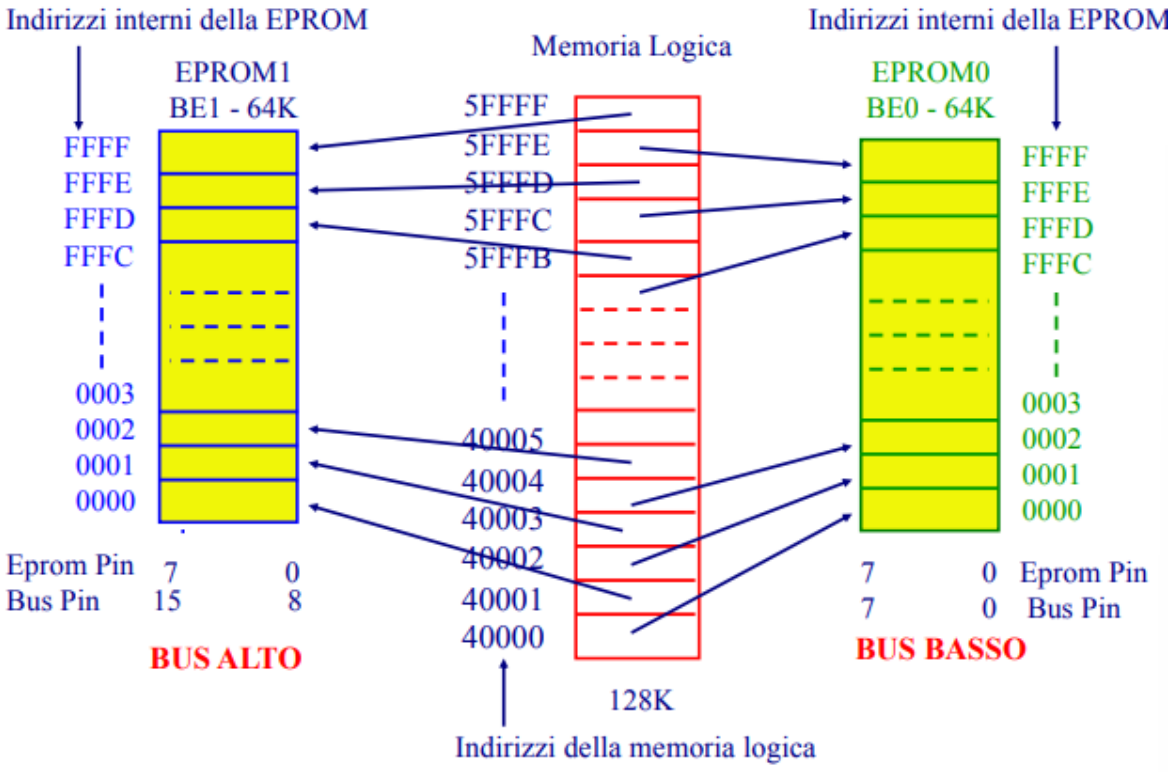
L'informazione non risiede nel solo indirizzo, ma nella nostra istruzione.



La decodifica si fa come se si avesse una memoria a 8 bit. Si usano dispositivi di taglia metà selezionati con BE0 e BE1. Le memorie vanno sempre in coppia.

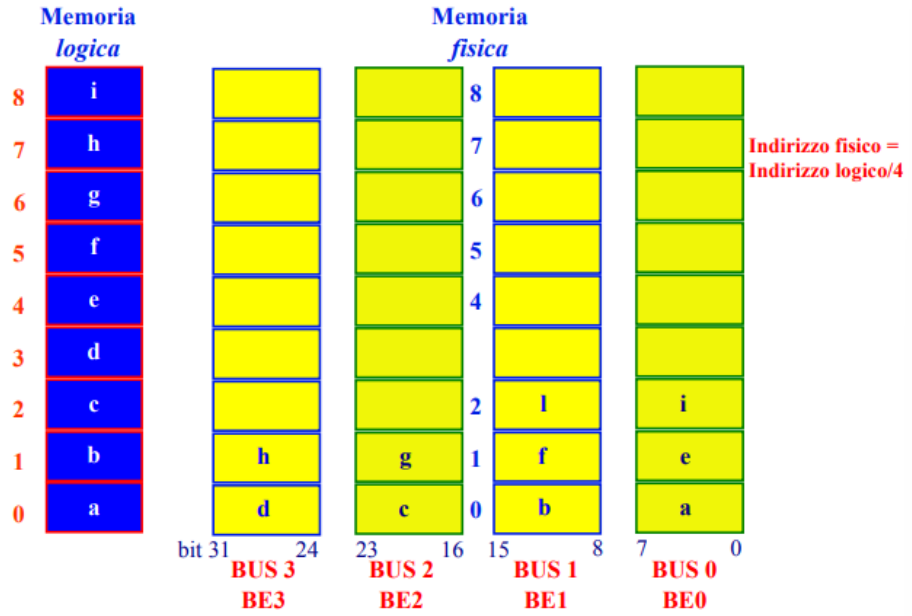
CSEPROM1 = BA19' x BA18 x BA17' x BE1

CSEPROM0 = BA19' x BA18 x BA17' x BE0



È errato mettere la prima memoria nei primi 64k e la seconda nei secondi 64k

MEMORIA CON PROCESSORI A PARALLELISMO 32 BIT



L'indirizzo fisico si ottiene dividendo l'indirizzo logico per 4 (cioè rimuovendo gli ultimi due bit).

Al posto di BA0 e BA1 ci saranno BE3, BE2, BE1, BE0.

Il grosso vantaggio è quello di poter trasferire, potenzialmente, il quadruplo delle informazioni.

Possiamo trasferire 32 bit (4 byte) in un unico ciclo di bus, solo se il dato è allineato alla taglia, cioè solo se lo trasferisco a un indirizzo che è multiplo di 4, compreso lo 0 (es. posso trasferire ABCD, ma non BCDE).

BE3	BE2	BE1	BE0	
1	1	1	1	Word 32 bit
0	0	1	1	Half word bassa
1	1	0	0	Half word alta
0	0	0	1	Byte 0-7
0	0	1	0	Byte 15-8
				etc.

Lo scambio fra i bytes (half word) dei banchi di memoria e i byte (half word) dei registri e viceversa avviene *all'interno* del microprocessore

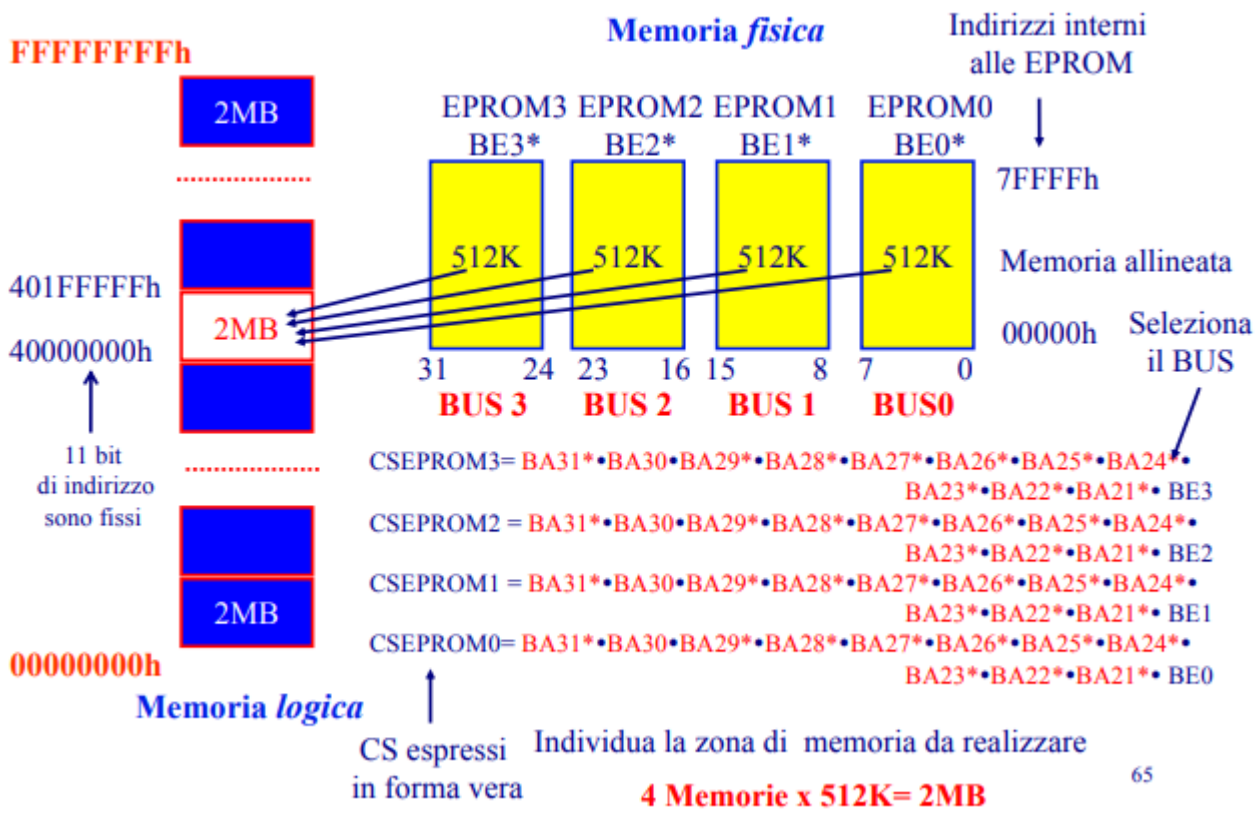
Se sono tutti 1 il trasferimento è di 4 byte (word), se sono attivi solo BE0 e BE1 oppure BE2 e BE3, possiamo trasferire 16 bit (half word), mentre se è attivo solo 1 il trasferimento è a 8 bit.

BE1BE2 =11 non è contemplato perché non ha senso voler memorizzare 16 bit a un indirizzo dispari.

I singoli byte possono essere trasferiti a qualsiasi indirizzo perché 1 byte è sempre allineato.

Con un bus dati a 32 bit le memorie hanno taglia ¼.

Se quando andiamo a dividere per 4 la taglia non esiste, dobbiamo dividere ulteriormente per 4.



I quattro chip select sono identici, a meno del bus enable BE.

LINGUAGGIO MACCHINA

L'insieme delle istruzioni e dei registri di una CPU costituisco l'Instruction Set Architecture (ISA).

Mediante l'ISA è possibile accedere alle risorse interne ed esterne. Tipicamente le istruzioni in linguaggio macchina sono generate da un compilatore, ma più raramente sono scritte dai programmatori. Purtroppo, quasi ogni CPU possiede un proprio ISA.

Esistono, in generale, due linee di pensiero:

- RISC: insieme ridotto di istruzioni semplici → molti registri interni
- CISC: insieme ampio di istruzioni complesse → pochi registri

Oltre alla possibilità di poter risolvere qualsiasi problema, un requisito fondamentale di un linguaggio macchina (ISA) è quello di minimizzare il tempo di esecuzione del codice.

Se CPI_{medio} è il numero medio di clock per l'esecuzione di un'istruzione, l'obiettivo è quello di minimizzare

$$CPU_{TIME} = N_{ISTRUZIONI} * CPI_{MEDIO} * T_{CK}$$

Lo stesso problema può essere quindi risolto con CPU_{TIME} diversi in base a:

- $N_{ISTRUZIONI}$ (RISC, richiedono in genere più istruzioni)
- CPI_{MEDIO} (RISC, tipicamente istruzioni più veloci)
- T_{CK} (Reti logiche semplici potenzialmente più veloci)

ISTRUZIONI E RISORSE INTERNE A UNA CPU

Le operazioni eseguibili da una CPU sono in genere molto più semplici delle istruzioni utilizzate nei linguaggi ad alto livello.

- Somme, sottrazioni, divisioni, moltiplicazioni, ecc.
- Letture e scritture in memoria e periferiche
- Confronto tra operandi
- Salti condizionati e incondizionati (avviene indipendentemente dall'esito di un confronto) ecc.

È possibile, mediante le istruzioni, accedere a risorse interne della CPU come registri architetturali e talvolta a registri di stato

REGISTRI DI UNA CPU

Ogni CPU possiede un certo numero di registri accessibili al programmatore. Il numero e la dimensione dei registri dipendono dall'ISA (e questo ha impatto sulla rete logica risultante). Ovviamente, avere molti registri general purpose (GP) è vantaggioso perché così si accede meno alla lenta memoria.

Avere istruzioni che possono usare tutti, o quasi, i registri GP senza vincoli, è vantaggioso.

Lo stesso ISA può essere realizzato con reti logiche completamente differenti. Queste reti hanno in genere prestazioni diverse.

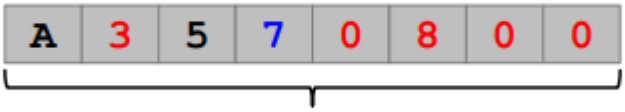
CODIFICA BINARIA DELLE ISTRUZIONI

Le istruzioni, per essere eseguite dalla rete logica CPU, devono essere codificate in binario secondo un formato noto e documentato dal produttore (datasheet). La codifica binaria deve contenere tutte le informazioni necessarie all'**unità di controllo** per poter eseguire l'istruzione.

Esistono CPU con codifica delle istruzioni a lunghezza:

- Costante (vantaggioso perché si conosce già l'indirizzo della prossima istruzione nel fetch)
- Variabile da istruzione a istruzione (negli x86 moderni, in realtà, queste istruzioni vengono trasformate in microistruzioni a lunghezza costante e poi eseguite)

```
Esempio:  LB  R7,0800(R3) - "Leggi un BYTE (8 bit)
all'indirizzo R3 + 0800h e trasferisci nel registro R7"
```



Ipotetica codifica dell'istruzione con 32 bit. I bit non utilizzati per codificare R3, R7 e 0800 rappresentano il *codice operativo* (*op code*)

LINGUAGGIO ASSEMBLER

La codifica delle istruzioni in linguaggio macchina è poco intuitiva per gli esseri umani. Nel linguaggio assembler si codificano le informazioni in modo (un po') più intuitivo.

Es. 014FA27Dh -> ADD R1, R2, R3 (R1 = R2 + R3)

Un altro significativo vantaggio è quello di poter definire delle label utili nei salti

```
LOOP:  SUB R1,R1,R3
       . . . . . .
       BNEZ (R1),LOOP   ; salta a LOOP se R1!=0
```

NOTAZIONE PER LA COSTRUZIONE DI CONFIGURAZIONI BINARIE

Consente di esprimere efficacemente configurazioni binarie:

- Traslazione logica a sinistra di n bit: << n (inserisce uno 0 a destra)
- Traslazione logica a destra di n bit: >>n (inserisce uno 0 a sinistra)
- Concatenamento di due campi: ##
- Ripetizione n volte di x: $(x)^n$
- Ennesimo bit di una configurazione binaria x: x_n (il pedice seleziona un bit)
- Selezione di un campo in una stringa di bit x: $x_{n\ldots m}$ (un range in pedice seleziona il campo)

Es. Data la configurazione binaria di 8 bit $C = 01101100_2$

- $C \ll 2$: 10110000_2
- $C_{3\ldots0}\#\#1111$: $1100|1111_2$
- $C^2_{3\ldots0}$: 11001100_2
- $(C_6)^4\#\#C \gg 4$: $1111|00000110_2$

- Trasferimento di un dato: $\leftarrow$

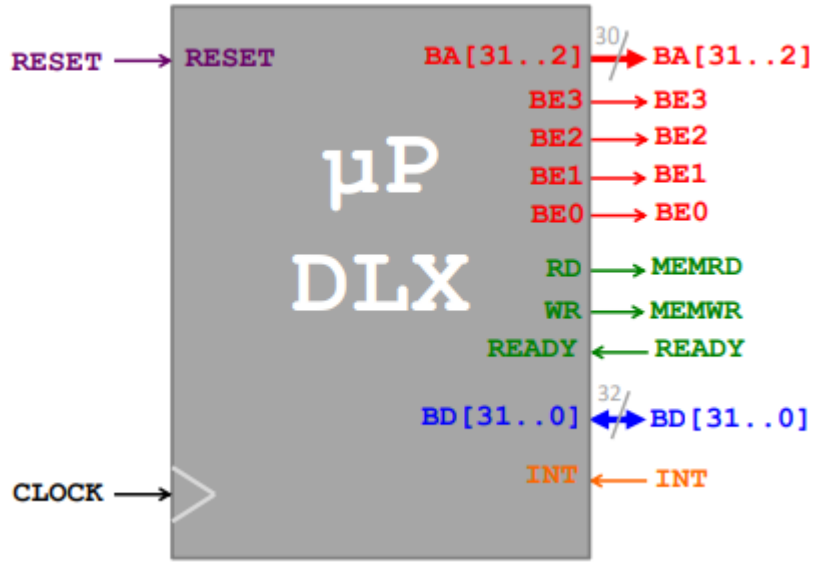
Il numero di bit trasferiti è dato dalla dimensione del campo destinazione; la lunghezza del campo va specificata secondo la notazione seguente tutte le volte che non è altrimenti evidente

- Trasferimento di un dato di n bit: $\leftarrow_n$
  Questa notazione si usa per trasferire un campo di n bit, tutte le volte che il numero di bit da trasferire non è evidente senza la relativa indicazione esplicita
- Contenuto di celle di memoria adiacenti a partire dall'indirizzo x: $M[x]$

Es. $R1 \leftarrow_{32} M[x]$

$indica\ il\ trasferimento\ dalla\ memoria\ verso\ il\ registro\ R1\ dei\ 4\ byte: M[x], \ldots M[x+3]$

SEGNALI DEL PROCESSORE DLX



Il segnale di RESET è asserito, all'avvio, da una rete esterna. Anche i segnali di READY, INT sono generati da reti esterne ma utilizzati durante il normale funzionamento.

Il segnale INT è il segnale che consente, dall'esterno, di avvisare il microprocessore che sarebbe necessaria la sua attenzione. Si indirizza il processore a eseguire determinate operazioni quando se ne manifesta la necessità.

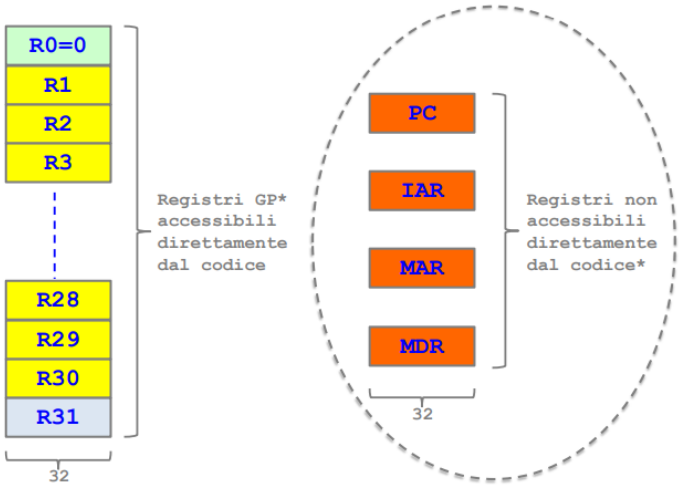
CARATTERISTICHE DELL'ISA DLX

- Unico spazio di indirizzamento di 4G
- 32 registri da 32 bit GP ($R_0, \dots, R_{32}, con\ R_0 = 0$)
- Istruzioni di lunghezza costante, 32 bit allineate
- Campi delle istruzioni di dimensioni/posizioni fisse
- Tre formati di istruzione: I,R,J
    - I: Istruzioni che utilizzano due operandi riferiti a registri e a un operando immediato da 16 bit con segno (che poi viene trasferito in un registro a 32 bit estendendolo in segno). Usate per l'accesso diretto alla memoria e al trasferimento dati verso/da i registri
    - R: Istruzioni che lavorano esclusivamente tra registri. Utilizzano un operatore a tre operandi riferiti a registri (32 bit di lunghezza). Usate per calcolo numerico su interi con la ALU
    - J: istruzioni di salto che contengono un operatore e uno spiazzamento da 26 bit con segno che, sommato all'indirizzo contenuto nel PC, fornisce indirizzo di destinazione del salto. Usate per implementare il controllo del flusso di un programma
- Non ci sono istruzioni per gestire lo stack

  Uno stack è un'area di memoria esterna che viene gestita col meccanismo di una pila. È importante perché ci consente di salvare l'indirizzo di ritorno quando chiamiamo una funzione (e anche per altri motivi).
- Per istruzioni che prevedono un indirizzo di ritorno (JAL/JALR), esso è salvato in R31
- Non esiste un registro di FLAG settato dalle istruzioni ALU. Le condizioni sono settate esplicitamente nei registri (istruzioni SET).

  Tipicamente dopo una SET si hanno istruzioni di salto condizionato.
- È presente un'unica modalità di indirizzamento in memoria (indiretto, mediante registro + offset)
- Le operazioni aritmetico/logiche sono eseguite solo tra registri (non tra registri e memoria)
- Esistono alcune istruzioni (MOV32I e MOVI23) per spostare dati tra registri GP e registri speciali e viceversa

REGISTRI DEL DLX



PC – Program counter

IAR – Interrupt Address Register

MAR – Memory Address Register

MDR – Memory Data Register

PC e IAR riguardano lo stato della macchina

Possiamo fare operazioni, ma con un vincolo: solo tra due registri oppure tra un registro e una costante. Non possiamo fare operazioni tra registri e memoria e uno dei valori non può stare in memoria. Non possiamo sommare un registro (es. R3) e un indirizzo. Se vogliamo fare operazioni con qualcosa che è in memoria dobbiamo caricare in registro quello che c'è in memoria e poi fare l'operazione tra i registri.

Ci sono delle istruzioni speciali per accedere, leggere, sovrascrivere su registri.

In ogni registro deve esserci un dato a 32 bit. Dati a un numero inferiore di bit devono essere estesi a 32 bit (mantenendo il segno e riempiendo di zeri i bit che mancano per arrivare a 32 bit).

Oltre ai registri base di tipo generale (GPR – General Purpose Register) del DLX ci sono anche altri registri non accessibili direttamente dal codice, tra cui:

- **PC – PROGRAM COUNTER**
  Registro della prossima istruzione da svolgere/su cui fare fetch (lettura ed esecuzione).
  Registro contatore di programma, contiene il puntatore all'indirizzo della prossima istruzione da eseguire.
- **IAR – INTERRUPT ADDRESS REGISTER**
  Registro di indirizzamento delle interruzioni.
  Utilizzato per la gestione delle interruzioni, in quanto contiene l'indirizzo di ritorno dalle routine di interrupt fetch in un'area di memoria (indirizzo zero) dove c'è la procedure per gestire interrupt.
  IAR memorizza il program counter nel momento in cui si interrompe il main e si va su interrupt.
  Un interrupt non può essere interrotto da un altro interrupt.

Ci sono altri registri che servono per l'esecuzione delle istruzioni: hanno a che fare con la memoria. Il programmatore non può leggerli e scrivere in modo esplicito, ma a volte vengono modificati, quando si accede alla memoria (load byte di qualcosa a un determinato indirizzo (MAR)).

- **MAR – MEMORY ADDRESS REGISTER**
  Registro di indirizzamento di memoria.
  Contiene l'indirizzo di memoria a cui accedere per la lettura o scrittura dei dati. Quando si accede alla memoria, contiene l'indirizzo al quale il DLX vuole accedere.
- **MDR – MEMORY DATA REGISTER**
  Registro di lettura e scrittura della memoria.
  Contiene i dati da leggere o scrivere in memoria, dati nell'indirizzo contenuto in MAR.
- **TEMP – TEMPORARY REGISTER**
  Registro temporaneo, a perdere. È disponibile per la memorizzazione dei dati intermedi durante l'esecuzione di alcune istruzioni del DLX.

TIPI DI DATO DEL DLX

Nel DLX sono disponibili tre tipi di dato:

- BYTE (8 bit)
- HALF-WORD (16 bit)
- WORD (32 bit)

I dati di dimensione inferiore a 32 bit letti in memoria devono essere estesi a 32 bit durante il caricamento dei registri.

Questa operazione può essere eseguita mantenendo o meno il segno del dato letto dalla memoria.

In molti casi è necessario estendere la rappresentazione di un dato codificato con n bit in un dato con una rappresentazione a m bit (m>n)

Es. Vogliamo trasferire un byte dalla memoria in un registro a 32 bit.

10110101.

Assumendo che il dato sia senza segno, l'estensione a 32 bit avviene aggiungendo 24 zeri

00000000000000000000000010110101 oppure (0)24##10110101

Se il dato è con segno, l'estensione avviene replicando 24 volte il bit di segno

11111111111111111111111110110101 oppure (1)24##10110101

SET DI ISTRUZIONI DEL DLX

- Le principali istruzioni aritmetiche e logiche:
  - Istruzioni logiche con op. immediato: AND, ANDI, OR, ORI, XOR, XORI
  - Istruzioni aritmetiche: ADD, ADDI, SUB, SUBI

  Le istruzioni con la I finale (immediato) indicano che l'istruzione avviene tra un registro e una costante, altrimenti avviene tra due registri
  - Istruzioni di shift: SLL, SRL, SRA

    Shift logico a sinistra e shift aritmetico a sinistra coincidono. SLL può generare overflow e non preserva il segno.

    Trascinando a destra di una posizione un registro e inserendo a sinistra sempre il bit del segno si mantiene il segno del dato mentre lo si divide successivamente per 2.
  - Istruzioni di SET CONDITION: Sx, con x={EQ – Equal, NE – Not Equal, LT – Lesser than, GT – Greater than, LE – Lesser/Equal, GE – Greater/Equal}
- Le principali istruzioni di trasferimento dati
  - Load byte signed/unsigned (LB, LBU), load half-word signed/unsigned (LH, LHU), load word (LW)
  - Store byte, store half-word, store word: SB, SH, SW
  - Copia un dato da un registro DP a un registro speciale e viceversa: MOVS2I, MOVI2S
- Le principali istruzioni di trasferimento del controllo
  - Istruzioni di salto condizionato (PC+4 relative): BNEZ (Salta se non è 0), BEQZ (Salta se il registro è 0)

- Istruzioni di salto incondizionato J: assoluto (con reg.) e PC-relative
  Per salto incondizionato intendiamo l'idea di saltare sempre, non se qualcosa è 0 o 1. Si salta a un indirizzo (assoluto o PC-relative). Nella Jump non torniamo più indietro
- Istruzioni di chiamata e procedura Jump and Link (JAL). L'indirizzo di ritorno viene automaticamente salvato in R31. JAL con registro e immediato (PC-Relative)
  JUMP AND LINK – Salto e torno all'istruzione successiva (R31) rispetto alla JAL.
  L'indirizzo di memoria successivo al JAL è memorizzato in R31 (tipica chiamata a funzione a cui segue ritorno al main).

!! Si chiama una funzione con la JAL e si torna alla funzione con JUMP R31.
Se è immediato c'è una costante, che avranno dimensione inferiore a 32 bit.

- Istruzioni di ritorno dalla procedura di servizio delle interruzioni: RFE
  RFE – Return from exception (legata a interrupt), fa tornare indietro da interrupt.

- Nella LOAD BYTE assumiamo che il byte abbia segno, nella LOAD BYTE UNSIGNED assumiamo che il byte non abbia segno. Sto leggendo un valore a 8 bit nello spazio di indirizzamento che devo inserire in un registro a 32 e devo dire come voglio che venga esteso.
- Nella STORE non è necessaria l'estensione perché andiamo a scrivere 8/16/32 bit. Devo sapere di che tipo è solo in lettura.

Data Transfer

```
LW   Ra,Imm16bit(Rb)
LB   Ra,Imm16bit(Rb)
LBU  Ra,Imm16bit(Rb)
LH   Ra,Imm16bit(Rb)
LHU  Ra,Imm16bit(Rb)
SW   Ra,Imm16bit(Rb)
SH   Ra,Imm16bit(Rb)
SB   Ra,Imm16bit(Rb)

MOVS2I   Ra,Rs*
MOVI2S   Rs*,Ra

Special register Rs* (IAR)
```

Aritmetiche/logiche

```
ADD    Ra,Rb,Rc
ADDI   Ra,Rb,Imm16bit
ADDU   Ra,Rb,Rc
ADDUI  Ra,Rb,Imm16bit
SUB    Ra,Rb,Rc
SUBI   Ra,Rb,Imm16bit
SUBU   Ra,Rb,Rc
SUBUI  Ra,Rb,Imm16bit
SLL    Ra,Rb,Rc
SLLI   Ra,Rb,Imm16bit
SRL    Ra,Rb,Rc
SRLI   Ra,Rb,Imm16bit
SRA    Ra,Rb,Rc
SRAI   Ra,Rb,Imm16bit
OR     Ra,Rb,Rc
ORI    Ra,Rb,Imm16bit
XOR    Ra,Rb,Rc
XORI   Ra,Rb,Imm16bit
AND    Ra,Rb,Rc
ANDI   Ra,Rb,Imm16bit
LHI    Ra,Imm16bit
```

Controllo

```
Sx     Ra,Rb,Rc
SxI    Ra,Rb,Imm16bit
BEQZ   Ra,Imm16bit
BNEZ   Ra,Imm16bit
J      Imm26bit
JR     Ra
JAL    Imm26bit
JALR   Ra
```

x può essere:
LT,GT,LE,GE,EQ,NE

Per le istruzioni aritmetiche: l'immediato a 16 bit è esteso senza segno se di tipo U (unsigned) altrimenti con segno.

Per istruzioni logiche, sempre estensione senza segno.

$Ra \in \{R0^{+},R1,..,R30,R31\}$

$Rb \in \{R0,R1,..,R30,R31\}$

$Rc \in \{R0,R1,..,R30,R31\}$

+Ra non può essere R0 come registro destinazione di istruzioni *load*, MOV2SI, *aritmetico/logiche*, LHI e SET

Il primo registro è quello di destinazione.

ISTRUZIONI ARITMETICO – LOGICHE

CENTRALI

- **ADD**
  Somma due registri Rb e Rc e memorizza il risultato nel registro Ra. Il primo registro è il registro di destinazione, mentre il secondo e il terzo sono le sorgenti.
- **ADDI**
  La somma avviene tra un registro Rb e un immediato (un valore a 16 bit). Nell'istruzione come immediato compare una costante a 16 bit.
  Il risultato verrà inserito dall'istruzione nel registro Ra.
  Bisogna estendere l'immediato da 16 a 32 bit (avviene automaticamente col DLX). Se l'operazione è aritmetica c'è l'estensione con il segno.
- **ADDU** – Add Unsigned con due registri come operandi sorgenti
- **ADDUI** – Add Unsigned con un registro e un immediato come sorgente
- **SUB** – Sottrazione con segno tra due registri
- **SUBI** – Sottrazione con segno tra registro e immediato a 16 bit
- **SUBU** – Sottrazione unsigned tra due registri
- **SUBUI** – Sottrazione unsigned tra un registro e immediato a 16 bit

ISTRUZIONI DI SHIFT (La destinazione è sempre un registro)

- Shift o con registro con immediato a 16 bit
- **SLL** – Shift logico a sinistra
- **SLLI** – Shift logico a sinistra con immediato
- **SRL** – Shift logico a destra
- **SRLI** – Shift logico a destra con immediato
- **SRA** – Shift aritmetico a destra
- **SRAI** – Shift aritmetico a destra con immediato

ISTRUZIONI LOGICHE (La destinazione è sempre un registro)

- **OR** – Con operandi due registri
- **ORI** – Con operandi un registro e un immediato
- **XOR** – Con operandi due registri
- **XORI** – Con operandi un registro e un immediato
- **AND** – Con operandi due registri
- **ANDI** – Con operandi un registro e un immediato

- **LHI** – Load High Immediate, sarà eseguita da un ADDI.
  Nonostante sia presente la parola LOAD, non è un'operazione di load dalla memoria. È un'operazione di creazione di un indirizzo in un registro piazza i 16 bit dell'immediato nella porzione più significativa del registro destinazione e riempie la parte restante con tutti zeri. Prende i 16 bit della costante/immediato e li mette in quelli più significativi. I meno significativi diventano tutti zero, questo perché il contenuto del registro deve avere 32 bit.
  È un modo rapido per creare delle costanti di 32 bit in un registro in cui ciò che dobbiamo impostare (con ADDI) sono gli ultimi 16 bit.

DATA TRANSFER

Tutte le istruzioni di questo tipo hanno la stessa struttura.

- **LOAD WORD**
  Stiamo leggendo qualcosa dallo spazio di indirizzamento (32 word).
  Bisogna mettere insieme l'informazione del registro sorgente Rb (a 32 bit) e un immediato a 16 bit (esteso a 32 bit). Il DLX estende il valore e lo somma al registro sorgente.
  Il risultato di questa somma è l'indirizzo nel quale viene detto il dato dell'istruzione (LW) quindi Load Word, e verrà inserito nel registro destinazione, ossia Ra.
  Sia di LOAD e di STORE, indipendentemente dalla taglia del dato che viene letto o scritto, per tutte queste istruzioni, il calcolo dell'indirizzo è sempre lo stesso (registro Rb a 32 bit + immediato a 16 bit esteso a 32 bit con segno).
  Es. LOAD WORD
  Supponiamo che il registro Rb sia R8 e che l'immediato sia uguale a zero.
  Dunque, questa istruzione sarà LW R8, 0(R18).
  Questa istruzione carica 32 bit a partire dall'indirizzo R18+0 e questi 32 bit vengono messi dentro il registro R8.
  L'indirizzo deve essere allineato, perciò guardiamo i due bit significativi della somma tra R18 e l'immediato se il contenuto di R18 non è divisibile per 4 (ultimi 2 bit a zero) il DLX genera eccezione (non va bene) e il sistema operativo deciderà cosa fare e verrà generata una sorta di interrupt.
  LOAD WORD non ha la versione unsigned perché WORD ha 32 bit, quindi, non è necessario estendere nulla.
- **LOAD BYTE** è sempre allineato
- **LOAD BYTE UNSIGNED** anche
- **LOAD HALFWORD** – L'indirizzo deve essere pari
- **LOAD HALFWORD UNSIGNED** – L'indirizzo deve essere pari
- **STORE WORD** – L'indirizzo deve essere multiplo di 4 – Gli ultimi due bit devono essere entrambi 0.
- **STORE HALFWORD** – L'indirizzo deve essere pari
- **STORE BYTE** – Prendo il byte meno significativo contenuto nel registro Ra e lo scrivo all'indirizzo ottenuto con registro sorgente + immediato a 16 bit esteso con segno a 32 bit. Il byte è sempre allineato quindi non ci sono mai problemi. Qualsiasi indirizzo va bene.

ISTRUZIONI DI CONTROLLO

SET x → Sx

X è una condizione – minore, maggiore, minore uguale, maggiore uguale, uguale, non uguale (LT, GT, GE, EQ, NE).
SNE – Set not Equal
L'operando di destinazione è un registro, non potrà mai essere R0 (perché contiene una costante) ed è meglio che non sia R31.
Nel caso delle SET gli operandi sorgenti possono essere:

- **Sx** – Due registri
- **SxI** – Un registro e un immediato a 16 bit

ISTRUZIONI DI SALTO – BRANCH INSTRUCTIONS

Ci sono due istruzioni di salto disponibili

- **BEQZ** – Branch Equal Zero
  Salta se un registro contiene il valore 0.
- **BNEZ** – Branch Not Equal Zero
  Salta se un registro non contiene il valore zero.

Prima dell'istruzione di salto c'è un'istruzione di SET → la condizione è separata dal salto.

L'indirizzo di destinazione è calcolato sommando all'immediato a 16 bit esteso in segno con segno e sommati al Program Counter + 4. L'indirizzo è PC-Relative. Il salto è fatto relativo a PC+4.

Si dice che "il branch è preso" se la condizione è vera. Il salto può essere preso o non preso.

ISTRUZIONI DI JUMP (Senza ritorno)

- **JR** – Salto assoluto
  Salto all'indirizzo contenuto in un registro, posso saltare in ogni punto dello spazio di indirizzamento (devo mettere nel registro indirizzo desiderato).
- **J**
  Versione con immediato. Istruzione PC-Relative, la costante è a 26 bit esteso a 32 bit e sommato a PC+4 → 6 bit sono dedicati al codice operativo

ISTRUZIONI DI JUMP AND LINK

- **JAL** – Jump and Link
  Salto con ritorno. In R31 viene inserito PC+4 (indirizzo di ritorno) che serve per ritornare. C'è immediato
- **JALR** – Jump and Link Register
  Salto all'indirizzo contenuto nel registro e poi link in R31 e torno all'indirizzo.

FORMATO DELLE ISTRUZIONI

Nel DLX abbiamo tre tipi di formati per le istruzioni: I, J, R. Le istruzioni sono tutte a 32 bit.

Nelle istruzioni c'è sempre un codice operativo base, di 6 bit. Il codice operativo è sempre nei primi 6 bit dell'istruzione.

Nei formati di istruzioni I ed R, se ci sono dei registri sono sempre nelle stesse posizioni.

Mentre la rete di decodifica lavora, posso già andare a trovare il registro sorgente, quindi le operazioni vengono fatte tutte velocemente.

Le istruzioni di tipo R sono le istruzioni aritmetico-logiche senza immediato, che sono anche le istruzioni più comuni (es. ADD, SUB). Le istruzioni aritmetico-logiche con immediato sono quelle di tipo I (es. ADDI, SUBI).

Nelle istruzioni di tipo R ci saranno sicuramente l'OPCODE e i tre registri (Due registri sorgente e un registro destinazione), che occupano in totale 21 bit (6 + 5 + 5 + 5). Possiamo usare gli 11 bit rimanenti come estensione dell'OPCODE per capire di che istruzione si tratta.

Nelle istruzioni di tipo I abbiamo i 6 bit di OPCODE, 5 bit di registro sorgente, 5 bit di registro destinazione e i 16 bit per l'immediato.

Le Branch sono anche istruzioni di tipo I.

Nelle istruzioni di tipo J ci sono i 6 bit dell'OPCODE e 26 bit dell'immediato, perché non abbiamo bisogno di controllare nessun registro. Le istruzioni J sono i salti incondizionati con e senza ritorno.

MODALITA' DI ACCESSO ALLA MEMORIA

Qualsiasi ISA mette delle istruzioni per poter leggere/scrivere all'interno dello spazio di indirizzamento.

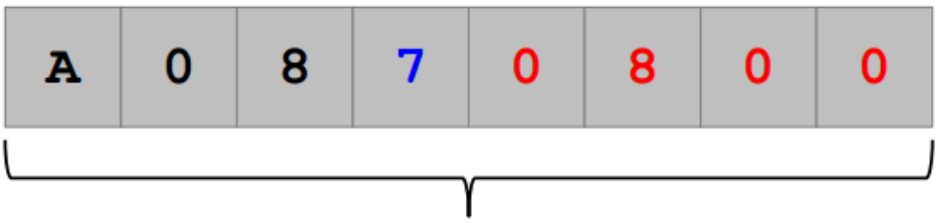
(INTEL ha due spazi di indirizzamento, uno per la memoria e uno per l'I/O; DLX ne ha 1).

Normalmente è possibile stabilire la dimensione del dato che può essere trasferito (BYTE, HALF-WORD, WORD).

INDIRIZZAMENTO DIRETTO

Con questa modalità l'istruzione contiene al suo interno un valore (cablato) che specifica l'indirizzo di accesso alla memoria.

Es. LB R7,0800h – "Leggi un BYTE all'indirizzo 0800h e memorizzala nel registro R7"



Ipotetica codifica dell'istruzione con 32 bit

INDIRIZZAMENTO INDIRETTO

L'indirizzo di accesso alla memoria è ottenuto sommando un valore costante presente nell'istruzione con il contenuto di un registro.

Indirizzo = costante + registro.

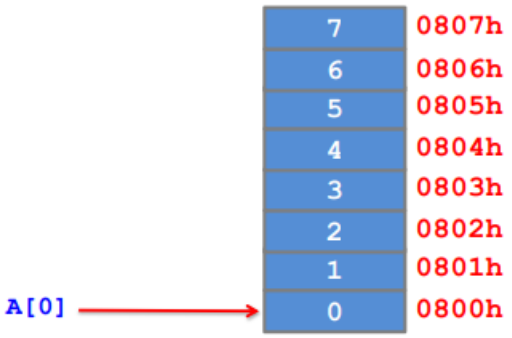
Il registro è cablato nell'istruzione ma il suo contenuto può cambiare a tempo di esecuzione.

Es. LB R7,0800(R3) – "Leggi un BYTE all'indirizzo R3 + 0800h e memorizzala nel registro R7"

Ipotetica codifica dell'istruzione con 32 bit

La differenza tra le due modalità di indirizzamento è notevole.

Es. “Sommare gli elementi di un array A di 8 elementi memorizzato a partire dall’indirizzo 0800h”




```
Diretto (non usato):

XOR  R8,R8,R8 ; R8=0

LBU R7,0800h
ADD R8,R8,R7 ; R8=R8+R7
LBU R7,0801h
ADD R8,R8,R7 ; R8=R8+R7
LBU R7,0802h
ADD R8,R8,R7 ; R8=R8+R7
LBU R7,0803h
ADD R8,R8,R7 ; R8=R8+R7
LBU R7,0804h
ADD R8,R8,R7 ; R8=R8+R7
LBU R7,0805h
ADD R8,R8,R7 ; R8=R8+R7
LBU R7,0806h
ADD R8,R8,R7 ; R8=R8+R7
LBU R7,0807h
ADD R8,R8,R7 ; R8=R8+R7
```

```
Indiretto:

        XOR  R8,R8,R8     ; R8=0
        ADDI R9,R8,8      ; R9=8

LOOP:   SUBI R9,R9,1      ; R9=R9-1
        LBU R7,0800h(R9)  ; legge BYTE
                          ; a 0800+R9
        ADD R8,R8,R7      ; R8=R8+R7
        BNEZ R9,LOOP      ; se R9!=0
                          ; salta a LOOP
```

- Il registro R9 è utilizzato in ogni iterazione per cambiare l’indirizzo base (0800h)
- Pensate se l’array fosse di 1000000 elementi...

Con l’indirizzamento diretto accedo a ogni indirizzo con un’istruzione diversa.

Di fatto si usa l’indirizzamento indiretto.

- XOR R8, R8, R8 – Fa lo XOR tra R8 e R8 (che viene 0) e memorizza in R8 il risultato. Quindi inizializza R8 a 0, perché R8 è l’accumulatore della somma dell’array
- ADDI R9, R8, 8 – Viene sommato a R8 la costante 8 e il risultato viene messo in R9, in cui finisce il valore 8 (8+0). R9 è l’indice dell’array e R8 è l’accumulatore.
  LOOP:
- SUBI R9, R9, 1 – Sottrae 1 a R9 e memorizza il risultato in R9. Ora in R9 c’è 7
- LBU R7, 0800h (R9) – Facciamo una lettura nello spazio di indirizzamento. Leggiamo un byte di tipo unsigned all’indirizzo 0800h + R9, cioè 08007h, che è di fatto l’ultimo elemento dell’array, e lo trasferiamo al registro R7. E’ inutile inizializzare R7 perché lo andiamo a sovrascrivere.
- ADD R8, R8, R7 – Andiamo ad aggiornare R8 sommando R8 + R7. Ora R8 contiene 0 + l’ottavo elemento dell’array, cioè 7. Quindi in R8 c’è 7.
- BNEZ R8, LOOP – Facciamo un salto per vedere se R9 è 0. Se R9 non è uguale a 0 vuol dire che non abbiamo scandito tutto l’array, perciò saltiamo a LOOP (ricomincia il LOOP). L’immediato di LOOP è -16, perché PC+4 punta all’istruzione successiva, ma io devo tornare all’inizio del LOOP, che è 4 istruzioni indietro.

Al prossimo ciclo del LOOP vado a prendere l’elemento “6” dell’array, viene aggiornato l’accumulatore e viene fatto il BRENCH. Il LOOP termina quando si arriva all’ultimo elemento dell’array (R9=0).

Il DLX prevede un’unica modalità di indirizzamento, cioè quello indiretto.

L’indirizzo a 32 bit è sempre ottenuto sommando un registro a 32 bit con un valore immediato a 16 bit esteso a 32 bit con segno.

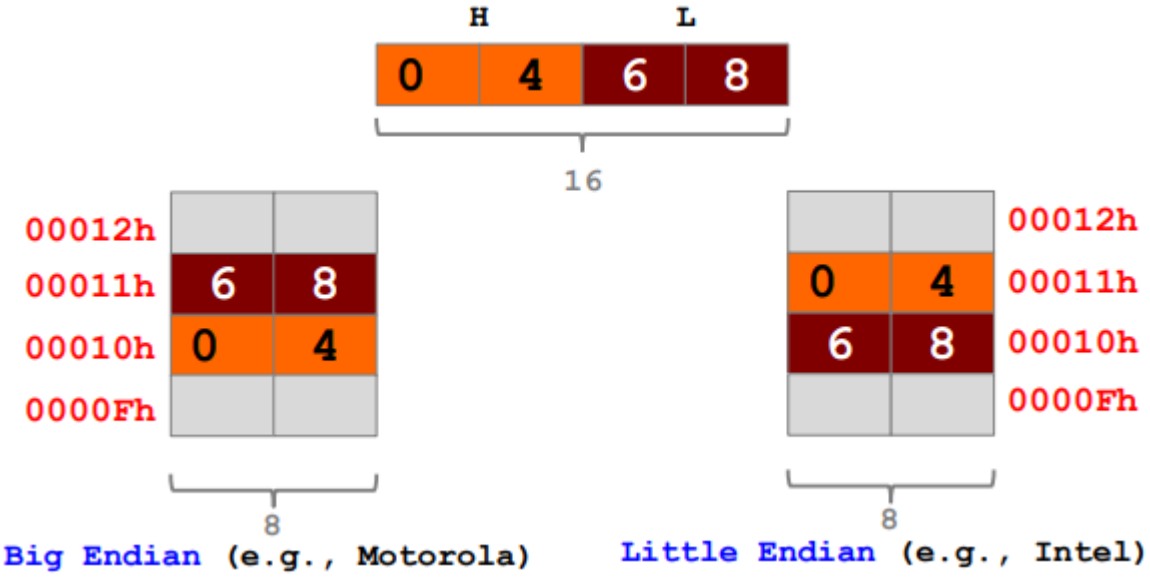
Es. LW R7, Imm_{16}(R8) – “Carica in R7, la word all’indirizzo ottenuto sommando a R8 il valore dell’immediato esteso a 32 bit con segno”:

$$R7 \leftarrow_{32} M[R8 + Imm_{16\_bit}[15]^{16} \#\# Imm_{16\_bit}[15..0]]$$

COME SONO MEMORIZZATI I DATI IN MEMORIA IN UN SISTEMA CON PARALLELISMO >8

Consideriamo un sistema con bus dati a 16 bit. Come possiamo memorizzare il valore a 16 bit 0468h a partire dall'indirizzo (che supponiamo a 20 bit) 0100h?

Esistono due convenzioni:



I due modi sono equivalenti, a patto che ci si metta d'accordo con chi utilizza il sistema.

ISTRUZIONI ARITMETICO LOGICHE

- Istruzioni a 3 operandi: due operandi sorgente e un operando destinazione.
- La destinazione è sempre un registro a 32 bit
- Le sorgenti possono essere due registri oppure un registro e un immediato a 16 bit.

Es. ADD R1, R2, R2 ; $R1 \leftarrow R2 + R3$ $Formato\ R$

Il fetch richiede un ciclo di bus, ma nella fase di esecuzione ci sono 0 cicli di bus perché tutto avviene all'interno della CPU

ADDI R1, R2, 3 ; $R1 \leftarrow R2 + 3$ $Formato\ I$

Il valore 3 dell'immediato a 16 bit è esteso a 32 bit

ISTRUZIONI DI SET CONDITION

Confrontano i due operandi sorgente e mettono a "1" oppure a "0" l'operando destinazione in funzione del risultato del confronto.

- "SET EQUAL" (SEQ, =) : setta se uguale
- "SET NOT EQUAL" (SNE, !=) : setta se diverso
- "SET LESSER THAN" (SLT, <) : setta se minore
- Ecc.

Gli operandi possono essere anche unsigned.

Es. SLT R1, R2, R3 ; $R1 \leftarrow 1$ se R2<R3, altrimenti $R1 \leftarrow 0$: formato R

SLTI R1, R2, 3 ; $R1 \leftarrow 1$ se R2<3, altrimenti $R1 \leftarrow 0$: formato I

ISTRUZIONI PER IL TRASFERIMENTO DATI

Sono istruzioni che accedono alla memoria (load e store): LB, LBU, SB, LH, LHU, SH, LW, SW

L'indirizzo dell'operando in memoria è la somma del contenuto di un registro a 32 bit con un "offset" di 16 bit esteso con segno a 32 bit.

L'istruzione è codificata secondo il formato I.

Es. LW R1, 40(R3) ; $R1 \leftarrow_{32} M[40 + R3]$

LB R1, 40(R3) ; $R1 \leftarrow_{32} (M[40 + R3]_7)^{24} \#\# M[40 + R3]$

LBU R1, 40(R3) ; $R1 \leftarrow_{32} (0)^{24} \#\# M[40 + R3]$

ISTRUZIONI PER IL TRASFERIMENTO DEL CONTROLLO:

SALTI INCONDIZIONATI

- "JUMP" : salto incondizionato
- "JUMP AND LINK" : salto incondizionato con ritorno

Es.

J offset ; $PC = PC + 4 + (offset[25])^6 \#\# offset$: formato J

JR R3 ; $PC = R3$: formato R

JAL offset ; $R31 = PC + 4$

; $PC = PC + 4 + (offset[25])^6 \#\# offset$: formato J

JALR R5 ; $R31 = PC + 4, PC = R5$

JAL R31 ; $PC = R31$

; Istruzione per tornare da una procedura

SALTO CONDIZIONATI (BRANCH)

E' possibile verificare solo due condizioni:

- "BEQZ" – "BRANCH EQUAL ZERO" : salta se registro è 0
- "BNEQZ" – "BRANCH NOT EQUAL ZERO" : salta se registro non è 0

Es.

BEQZ R4, Imm_{16} ; $se\ R4 = 0, PC = PC + 4 + Imm_{16}[15]^{16} \#\# Imm_{16}$

$altrimenti\ PC = PC + 4$

BNEZ R4, Imm_{16} ; $se\ R4 \neq 0, PC = PC + 4 + Imm_{16}[15]^{16}\#\#Imm_{16}$

$altrimenti\ PC = PC + 4$

Con una istruzione di tipo set seguita da un'istruzione di branch si realizza la funzione di compare and branch (confronto e salto condizionato dal risultato del confronto) senza bisogno di flag dedicati.

GENERARE VALORI A 32 BIT

Nel DLX è presente l'istruzione LHI ("Load High Immediate") che consente di creare rapidamente valori a 32 bit.

$LHI\ \ Rd, Imm_{16}$; $Rd = Imm_{16}\#\#0000h$

Inserisce in Rd il valore dell'immediato nei 16 bit più significativi e 0 nei rimanenti bit.

Tipicamente, LHI è utilizzata per generare indirizzi a 32 bit partendo da immediati a 16 bit.

INTERRUZIONI

In un sistema a microprocessore è di fondamentale importanza poter gestire eventi che si verificano all'esterno (ma non solo) della CPU. Per esempio, determinare se è stato premuto un tasto sulla tastiera, se il mouse è stato spostato etc.

Non c'è alcun sincronismo tra gli eventi che accadono all'esterno del processore e il clock. Gli eventi possono verificarsi in qualsiasi istante in modo totalmente asincrono.

Una strategia, poco efficiente, per raggiungere questo scopo consiste nel controllare periodicamente se tali eventi si sono verificati (**polling**). Questo può essere fatto interrogando di continuo la periferica che si desidera gestire. Ovviamente, con questa strategia, la CPU spende molti cicli macchina per la verifica. La CPU viene impiegata per compiere continuamente un'operazione che la maggior parte delle volte dà come risultato "non è successo nulla".

Una strategia molto più efficiente, basata su una strategia "push", consiste nell'uso di **interrupt.** Non ci si chiede se si è verificato l'evento, ma è la rete che lo gestisce che notifica direttamente se l'evento si è verificato.

Un **interrupt** è un evento che interrompe la CPU durante il regolare flusso di esecuzione del codice.

L'interrupt segnala che si è verificato un evento che merita immediata attenzione da parte della CPU.

La CPU ha bisogno di capire chi ha generato l'interrupt. Perciò bisogna prevedere un meccanismo col fine di comprendere chi ha generato l'interrupt e come gestirlo.

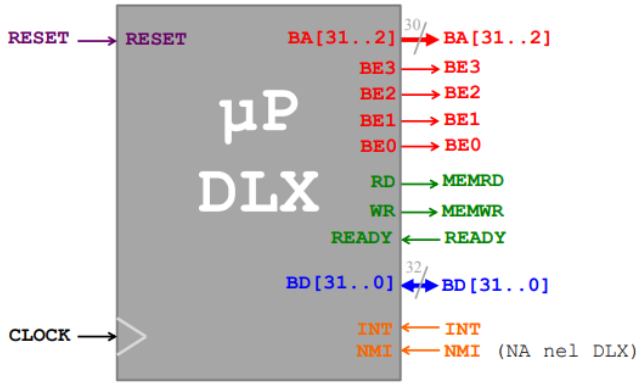
Se la CPU è abilitata a ricevere tale segnale, esegue automaticamente una porzione di codice denominata **interrupt handler** al fine di gestire l'evento.

Gli eventi possono essere relativi a fattori esterni o interni (es. divisione per zero, overflow …). Quando dipendono da fattori interni si parla di **eccezioni (exceptions).**

Inoltre, è possibile invocare l'handler mediante opportune istruzioni (es. per invocare **system call**).

In molte CPU c'è la possibilità di chiamare un interrupt via software, così da lanciare l'interrupt handler corrispondente anche se l'interrupt non è realmente avvenuto.

In ogni processore è presente almeno un segnale denominato **INT** per gestire le interruzioni. In molti casi, ma non nel DLX, è presente anche un ulteriore segnale denominato **NMI** per gestire interruzioni che non possono essere ignorate.



Nel caso in cui arrivino due interrupt in contemporanea l'handler può gestirli contemporaneamente o uno dopo l'altro in base alla loro priorità.

La CPU normalmente svolge operazioni utili ed è avvisata solo quando si verifica l'evento.

La pressione del tasto innesca l'esecuzione del codice dell'interrupt handler. Dal punto di vista software, se si stanno eseguendo delle istruzioni, l'istruzione viene portata a termine prima di eseguire l'interrupt handler. Assumeremo sempre che l'esecuzione dell'istruzione durante la quale si verifica l'interrupt sia sempre portata a termine prima di eseguire l'handler.

Il DLX è sensibile al segnale a livello dell'interrupt.

Esistono CPU sensibili al livello del segnale di interrupt, altre al fronte di salita, altre a entrambe le cose.

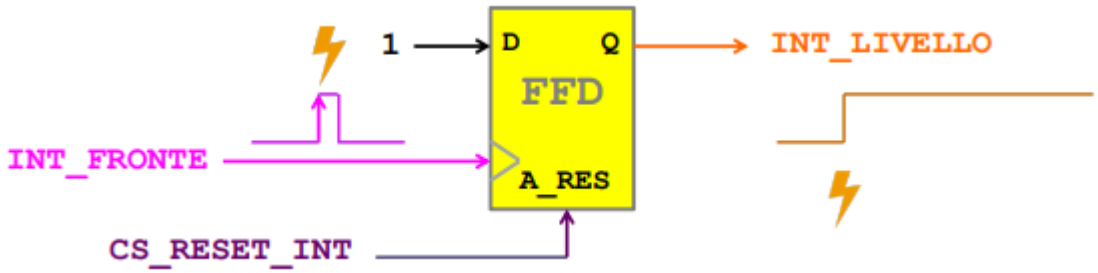
Nel caso del DLX assumeremo che la CPU sia sensibile al livello del segnale (1 se l'interrupt è attivo, 0 al contrario).

Nel caso dei dispositivi che generano interrupt, assumeremo che esso rimanga 1 fintantoché la causa che lo ha generato non sia stata gestita dalla CPU.

Pertanto, se una periferica ha un interrupt a livello asserito, rimane tale fintantoché l'interrupt non è gestito dalla CPU (non necessariamente subito).

In alcuni casi, nell'handler può essere necessario eseguire delle operazioni software per poter portare al livello logico 0 il segnale di interrupt proveniente dall'esterno dopo aver gestito l'evento.

Per trasformare il segnale INT da fronte a livello basta un FF-D. Il livello logico del segnale INT_LIVELLO deve essere portato a 0 da un opportuno comando software (CPU) che asserisce il segnale CS_RESET_INT



GESTIONE DI INTERRUZIONI MULTIPLE

Escludendo NMI il DLX ha un solo segnale di interrupt denominato INT. Per gestire, come accade tipicamente, interruzioni derivanti da sorgenti multiple, si convogliano tutti gli interrupt verso l'unico segnale INT presente nel DLX.

Per determinare quale/i interrupt sono asseriti in un determinato istante è tipicamente necessario poter determinare lo stato delle richieste di interrupt mediante opportune istruzioni software.

Esistono anche delle reti, denominate PIC, che possono agevolare questo compito alla CPU.

In un sistema nel quale è presente più di una sorgente di interruzione è fondamentale poter associare un livello di priorità a ciascuna interruzione. Sarebbe auspicabile poter interrompere l'interrupt handler in esecuzione se giunge una richiesta di interruzione più prioritaria (annidamento). Nel DLX non è possibile.

INTERRUPT NEL DLX

Assumeremo che il DLX sia sensibile al livello del segnale di interrupt INT e non al suo fronte.

L'indirizzo di ritorno (PC+4) è salvato in IAR. In seguito all'arrivo di un interrupt, l'istruzione in corso è completata ed è eseguito il codice all'indirizzo 0x000.

Il ritorno dell'interrupt handler ($PC \leftarrow IAR$) avviene mediante l'istruzione RFE (Return From Exception).

In genere, ma non nel DLX base, gli interrupt possono essere abilitati o disabilitati mediante istruzioni. Nell'ISA DLX, è gestito un solo indirizzo di ritorno. Pertanto, il DLX disabilita le interruzioni mentre esegue l'handler e le riabilita automaticamente ritornando dall'handler (RFE). In caso contrario, nel DLX, servirebbe uno stack software.

Questo avviene perché nel DLX non è possibile annidare le istruzioni, perciò anche se arriva un'istruzione più prioritaria questa non viene considerata fino a quando non si finisce di gestire quella precedente.

Con annidamento (nesting) delle interruzioni si intende la possibilità di poter avviare un interrupt handler durante l'esecuzione di un altro handler. Questa caratteristica è standard nella maggior parte delle CPU in commercio, ma non è prevista nel DLX base.

Per poter annidare gli interrupt sarebbe necessario uno stack software (utilizzando l'istruzione MOVS21) e avere la possibilità di ri-abilitare gli interrupt nell'handler mediante opportune istruzioni (ENI) non previste dall'ISA base.

In caso multiple sorgenti di interruzione, nasce il problema di come associare una scala di priorità alle interruzioni. A tal fine esistono varie politiche: priorità fissa, variabile ecc. Ovviamente la priorità è cruciale non solo quando è possibile annidare gli interrupt.

INTERRUPT HANDLER E CONSISTENZA DEI DATI

Le richieste di interrupt possono verificarsi in qualsiasi momento. E' però necessario mantenere la consistenza dei dati in modo che il codice in esecuzione non sia modificato dall'arrivo o meno degli interrupt e di conseguenza dall'esecuzione o meno degli interrupt handler.

Per questa ragione è necessario fare in modo che l'interrupt handler non interferisca con il codice del programma (main) in esecuzione.

Ciò è possibile salvando e ripristinando i registri modificati dall'interrupt handler all'interno dello stesso codice (handler).

INTERRUPT HANDLER CON SINGOLA ISTRUZIONE

Nel caso di una singola sorgente di interruzione, il codice di un tipico interrupt handler potrebbe avere la struttura seguente:

```
0x0000      ;   Istruzioni che salvano i registri modificati nelle seguenti istruzioni
            ;   Codice di risposta della richiesta di interruzione
            ;   Istruzioni di ripristino dei registri modificati in precedenza
```

0xXXXX RFE ; Riorno dall'interrupt

INTERRUPT HANDLER CON MULTIPLE INTERRUIONI

0x0000 ; Istruzioni che salvano i registri modificati nelle seguenti istruzioni

; Codice di risposta della richiesta di interruzione

; Istruzioni di ripristino dei registri modificati in precedenza

0xXXXX RFE ;

0xZZZZ ; Salva registri – Codice Handler_1 – Ripristina registri e RFE

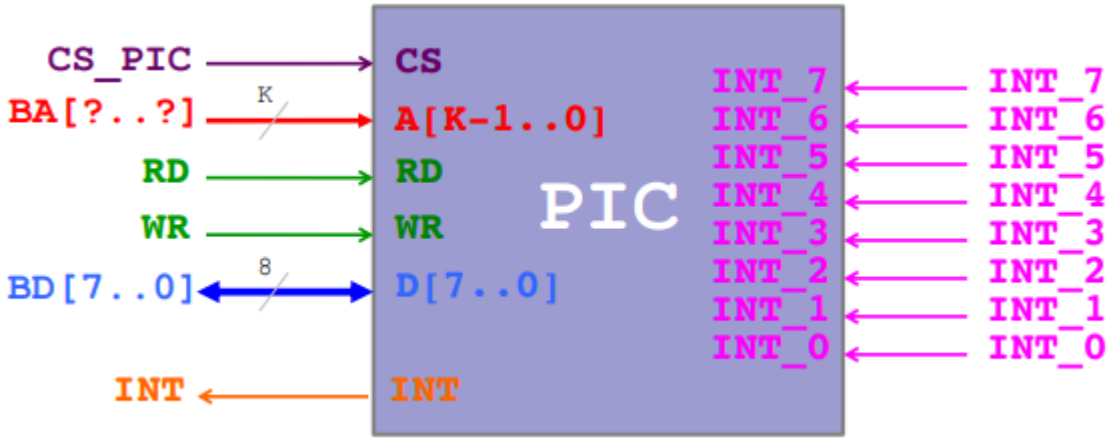
0xYYYY ; Salva registri – Codice Handler_2 – Ripristina registri e RFE

PROGRAMMABLE INTERRUPT CONTROLLER (PIC)

Con la strategia mostrata prima è il software, interrogando ogni singola periferica, a dover determinare qual è l'interrupt più prioritario.

A tal fine, sarà anche necessaria una opportuna infrastruttura hardware. Tuttavia, è possibile velocizzare e semplificare le reti logiche di supporto a questo compito mediante l'utilizzo di un dispositivo ad hoc (PIC). Il PIC si occupa di gestire multiple sorgenti di interruzione e di fornire direttamente alla CPU (su richiesta) qual è il codice/indirizzo dell'interrupt più prioritario tra quelli asseriti al momento.

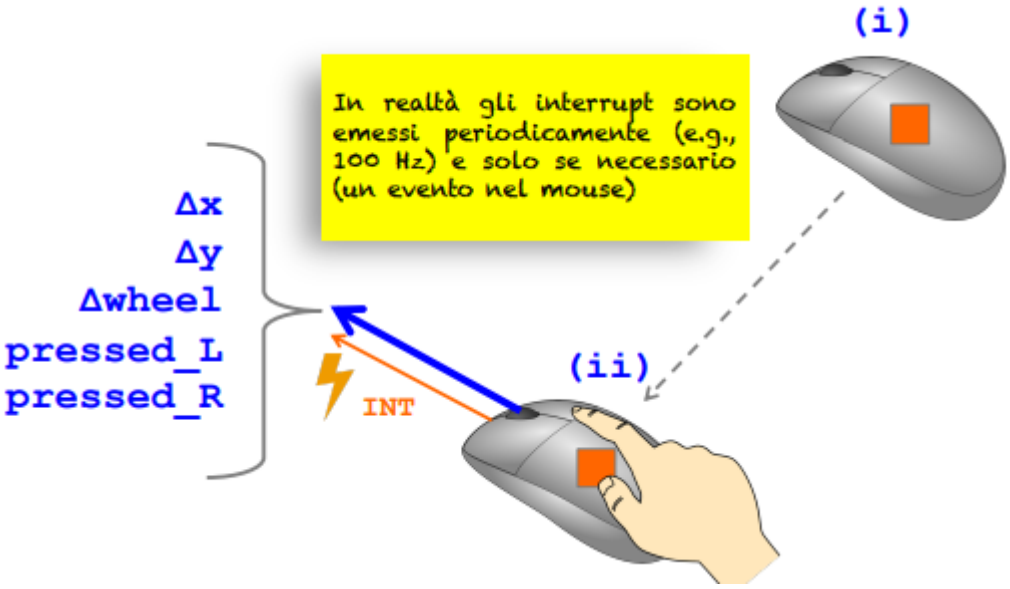
Tipicamente, in un PIC è possibile disabilitare le singole sorgenti di interruzione e stabilire il livello di priorità di ciascuna in accordo a varie politiche.



Le varie sorgenti di interruzione INT[7…0] sono inviate al PIC che si occupa di inviare la richiesta sull'unico pin INT del DLX.

Il PIC invia il segnale INT e fornisce alla CPU, su richiesta, il codice dell'interrupt con priorità più elevata tra quelli asseriti in quel momento.

Nel PIC c'è anche il segnale di WRITE per programmare la politica di gestione degli interrupt



In realtà le informazioni sono convogliate su un canale seriale (USB, PS/2) per ridurre il numero di connessioni/fili.

Tuttavia, possiamo pensare per le nostre finalità che l'interfaccia mouse/CPU esponga i segnali di una porta I/O standard (CS, RD, WR, D[7…0], indirizzi)

INTERRUPT NON MASCHERABILI

In una CPU (ma non nel DLX) può essere presente un ulteriore segnale (in input) denominato NMI (Not Maskable Interrupt).

A tale segnale sono collegate un numero limitato di sorgenti di interruzioni particolarmente critiche. Per esempio, l'output di una rete che rileva e segnala una imminente perdita di alimentazione elettrica.

Una richiesta di interrupt inviata sul pin NMI non può essere ignorata (eventuali istruzioni che disabilitano gli interrupt non agiscono per questo segnale) e interrompe l'esecuzione di altri handler.

Il segnale NMI va usato con cautela e solo per segnalazioni critiche alla CPU. Nel caso del DLX utilizzeremo solo INT.

Se fosse disponibile, per la gestione del segnale NMI sarebbe necessario inserire le istruzioni nella prima parte del preambolo all'indirizzo 0x0000, prima di gestire gli interrupt che sono inviati attraverso INT.

HANDSHAKE

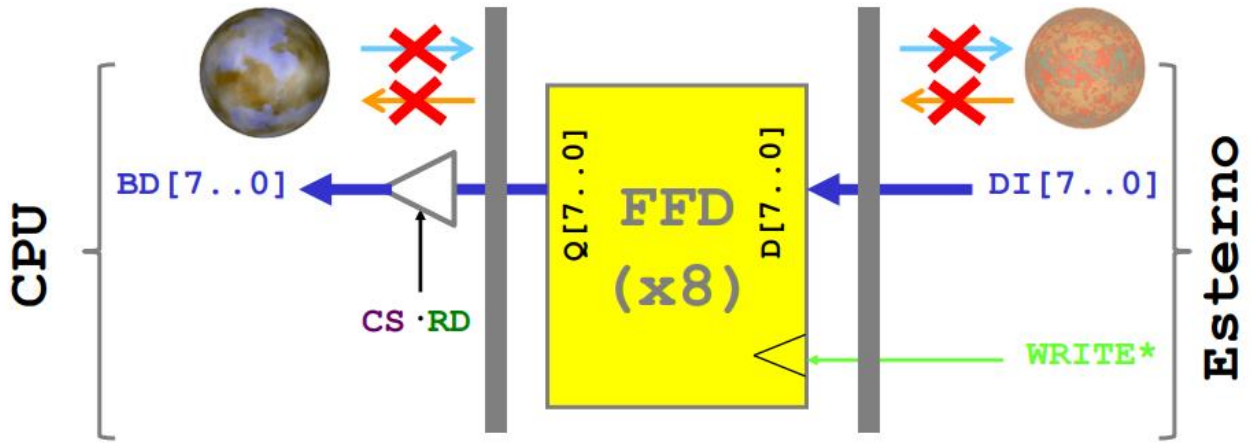
PORTE DI INPUT/OUTPUT

In precedenza, abbiamo visto come progettare delle semplici periferiche di I/O, per scambiare dati tra CPU e mondo esterno mediante un buffer.

Tuttavia, non vi era nessuna garanzia sul corretto esito dei trasferimenti. Infatti, cosa accade se, mentre la CPU scrive nella porta di output, un dispositivo esterno legge dalla medesima porta? Inoltre, cosa accade se la CPU legge un dato che in realtà non è mai stato scritto da un dispositivo esterno? Come può saperlo? Per questo, i trasferimenti sono intrinsecamente esposti a errori.

Inoltre, la gestione dei trasferimenti era totalmente a carico della CPU (che potrebbe fare altro). La CPU è molto veloce, mentre il dispositivo esterno in molti casi è molto lento, quindi la CPU potrebbe svolgere altre azioni.

Le porte I/O fino ad ora non erano in grado di generare interrupt con tutte le problematiche che ne derivano.

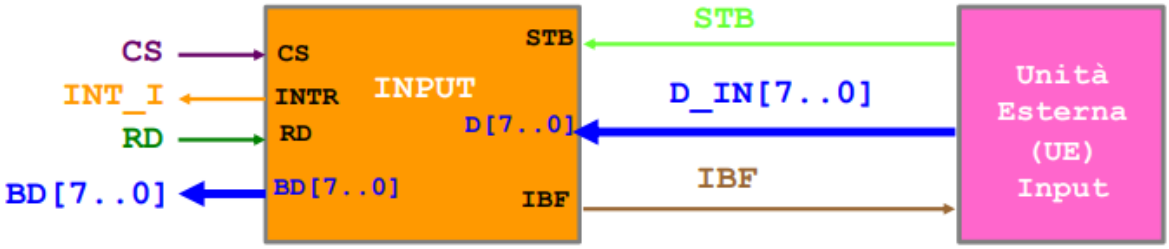


Con questa soluzione, sorgono degli evidenti problemi:

Come può sapere la CPU che è disponibile un nuovo dato scritto dall'esterno nella porta? Come si può sapere dall'esterno che la CPU ha letto il dato scritto in precedenza nella porta?

Questi problemi possono essere risolti in modo semplice attraverso la sincronizzazione tra le due entità. Per questo scopo l'handshake è un approccio semplice, efficiente e ampiamente utilizzato

SEGNALI DEL PROTOCOLLO HANDSHAKE(INPUT)

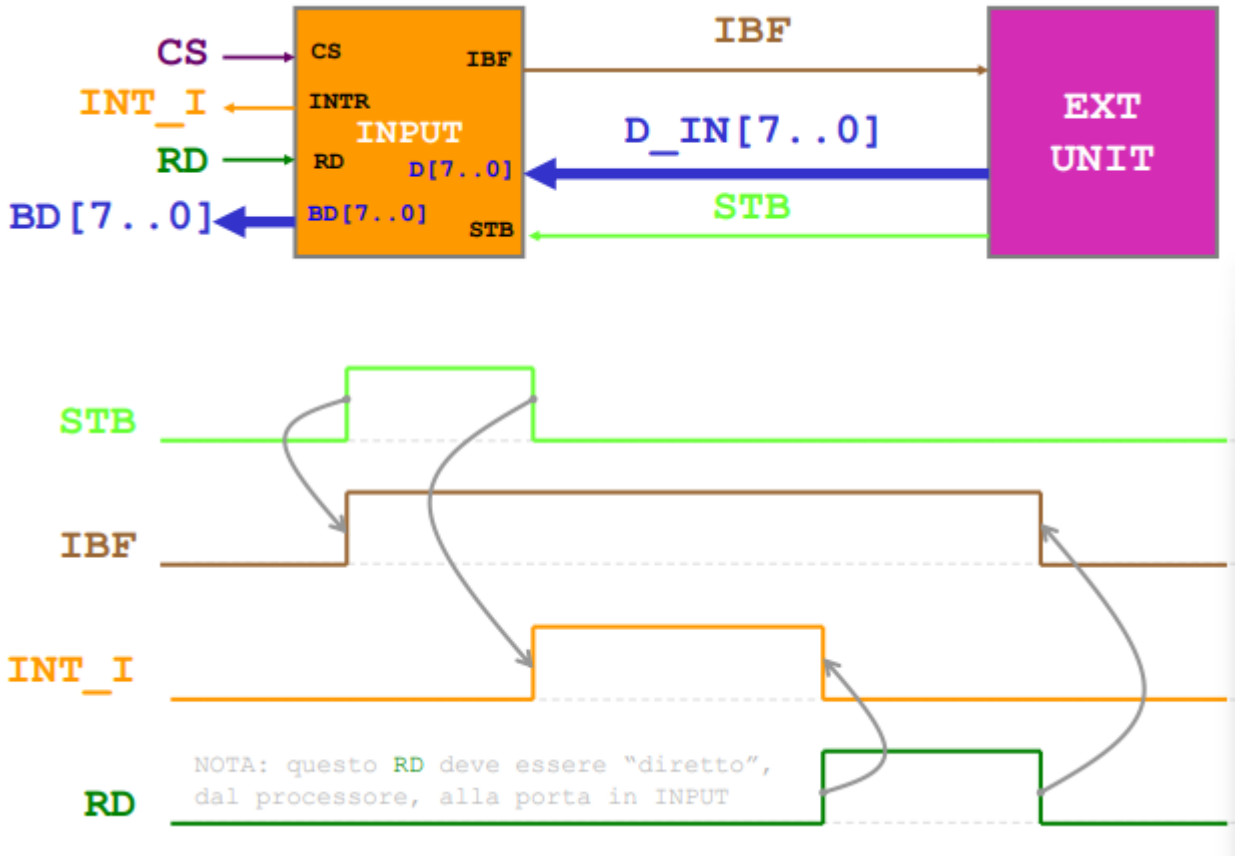


A sinistra c'è la CPU.

- Se IBF = 0 (INPUT BUFFER FULL), quando possibile UE può scrivere il dato nel buffer d'ingresso della porta. IBF dice se il BUFFER è pieno.
- Il segnale STB è equivalente al segnare di WRITE.
- UE, portando STB a 1, scrive il dato nella porta che contemporaneamente asserisce IBF.
- Quando UE porta STB a 0 (scrittura terminata), l'interfaccia attiva INT_I (Interrupt Request). INT_I=1 comunica che dall'esterno è stato scritto un nuovo dato, quindi il buffer si è riempito e INT_I comunica alla CPU che è finita la scrittura e sarebbe opportuno che la CPU andasse a leggere i dati scritti.

- Quando possibile, la CPU andrà a leggere il dato scritto nella porta da UE. Al termine, IBF andrà a zero (mentre INT_I va a 0, dall'inizio della lettura).

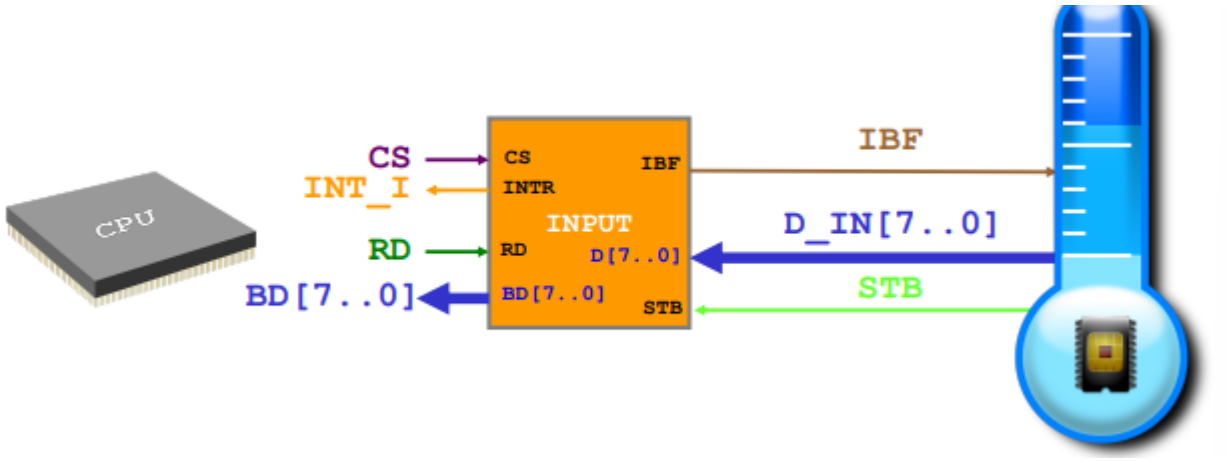
FORME D'ONDA



Per esempio, l'unità esterna potrebbe essere un sensore di temperatura.

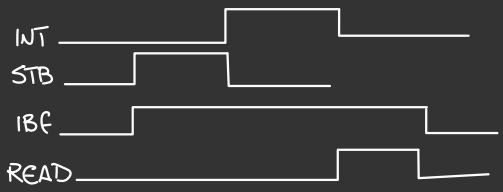
Il sensore invia i dati alla CPU attraverso la periferica di input quando una nuova misura è disponibile e IBF=0. Al termine di ogni scrittura nella porta da parte del sensore di temperatura, il segnale INT_I si asserisce.

Il sensore deve contenere una semplice rete logica in grado di gestire il protocollo di handshake.



SEGNALI DEL PROTOCOLLO HANDSHARE – OUTPUT

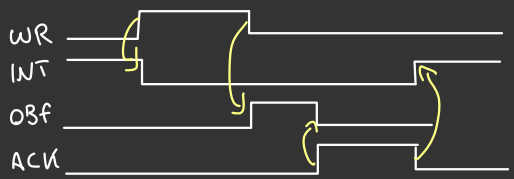
STB ↑ => IBF ↑
STB ↓ => INT ↑
INT ↓ => READ ↑
READ ↓ => IBF ↓



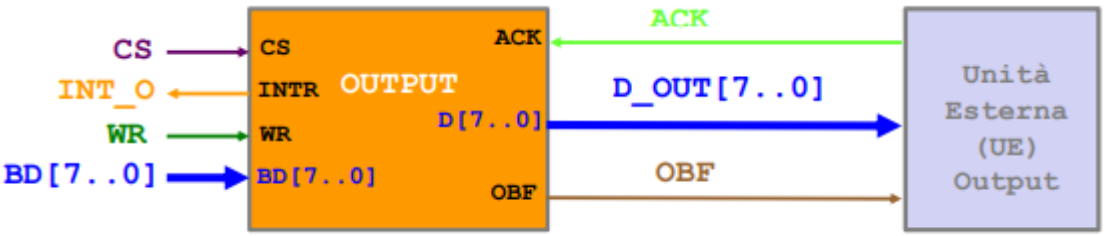
Porta scritta => Porta piena
Fine scrittura => Interrupt
Interrupt eseguito => lettura
Dato letto => Buffer vuoto

OUTPUT
OBF (Buffer pieno)
ACK (Dato scritto)

WR ↑ => INT ↓
WR ↓ => OBF ↑
ACK ↑ => OBF ↓
ACK ↓ => INT ↑

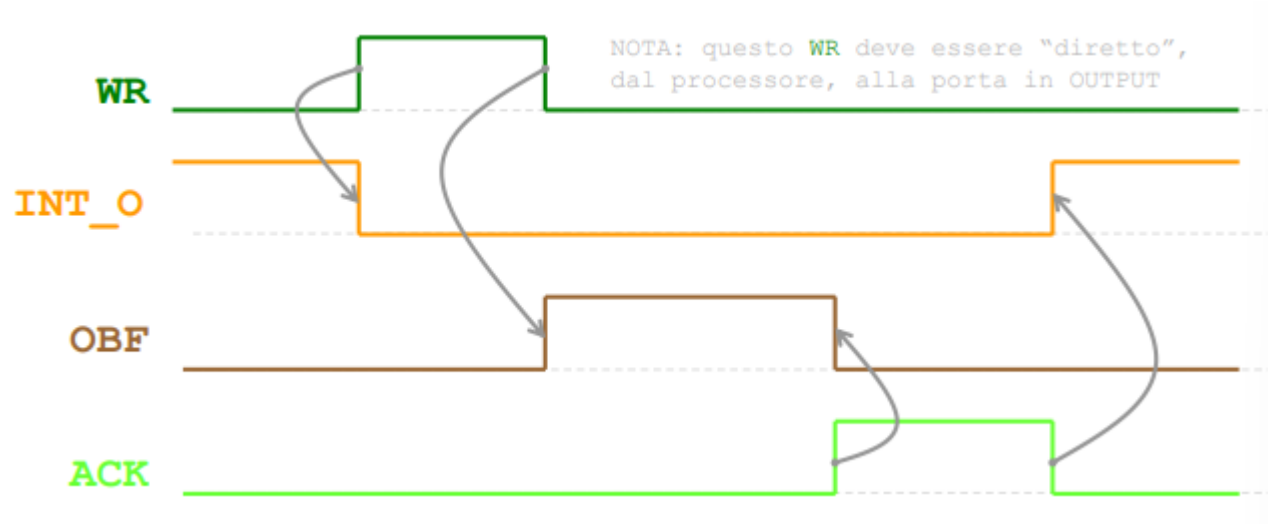


Scrivendo => Int = 0 (porta non può ricevere dati)
Scritto => Buffer pieno
Unità esterna legge buffer => Buffer vuoto
Unità esterna finisce di leggere i dati => Porta può ricevere dati



- Il segnale INT_O asserito comunica alla CPU che la porta può accettare un nuovo dato. Se INT_O è 1 allora l'unità esterna ha letto il dato che c'era prima.
- In risposta alla richiesta di interrupt, la CPU scrive, quando possibile, il dato sul buffer della porta.
- L'interfaccia segnala a UE che è disponibile un nuovo dato attivando OBF (Output Buffer Full).
- Quando possibile, UE legge il dato scritto dalla CPU asserendo ACK (Acknowledge).

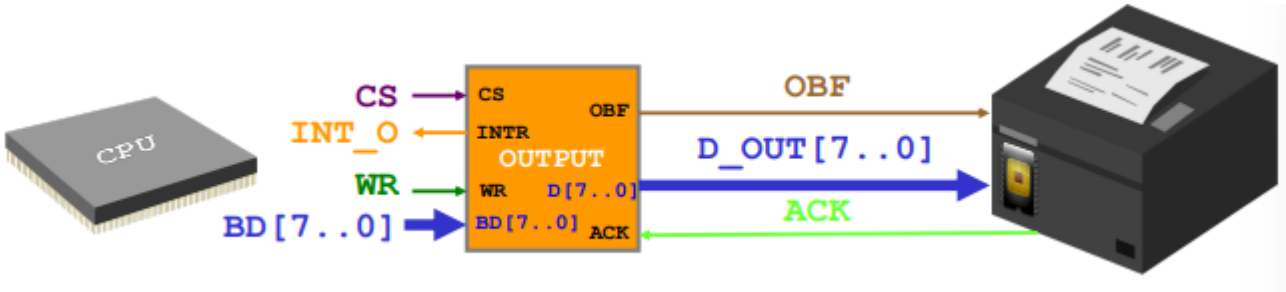
FORME D'ONDA



Un esempio di unità esterna in output potrebbe essere rappresentato da una stampante che imprime sulla carta un carattere alla volta.

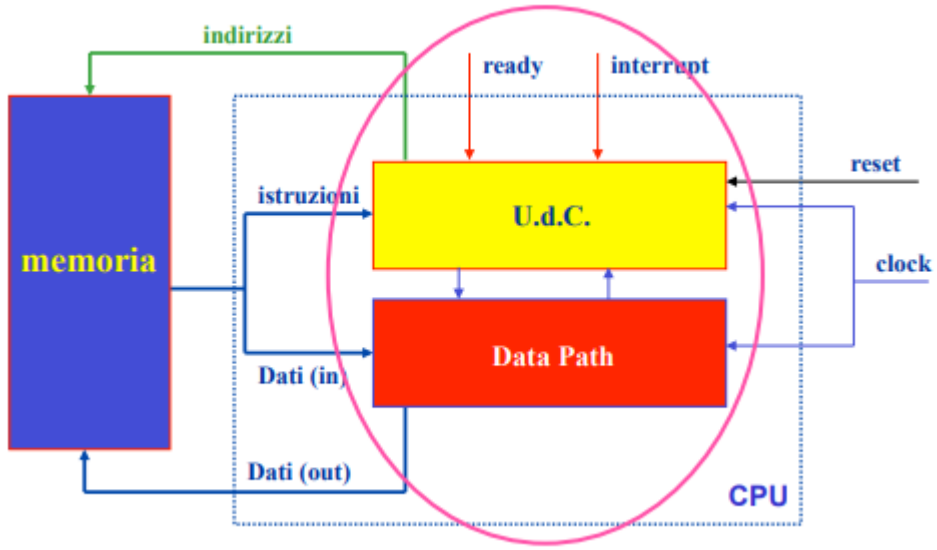
La CPU fornisce i dati alla stampante attraverso la periferica di output quando il segnale INT_O è asserito (questo implica che OBF sia 0). La stampante legge il dato solo quando il segnale OBF è asserito (per esempio, quando la porta in output comunica alla stampante che un nuovo dato è stato scritto dalla CPU nel buffer ed è quindi disponibile).

La stampante deve contenere una semplice rete logica in grado di gestire il protocollo di handshake.



DLX SEQUENZIALE

DATAPATH E UNITA' DI CONTROLLO



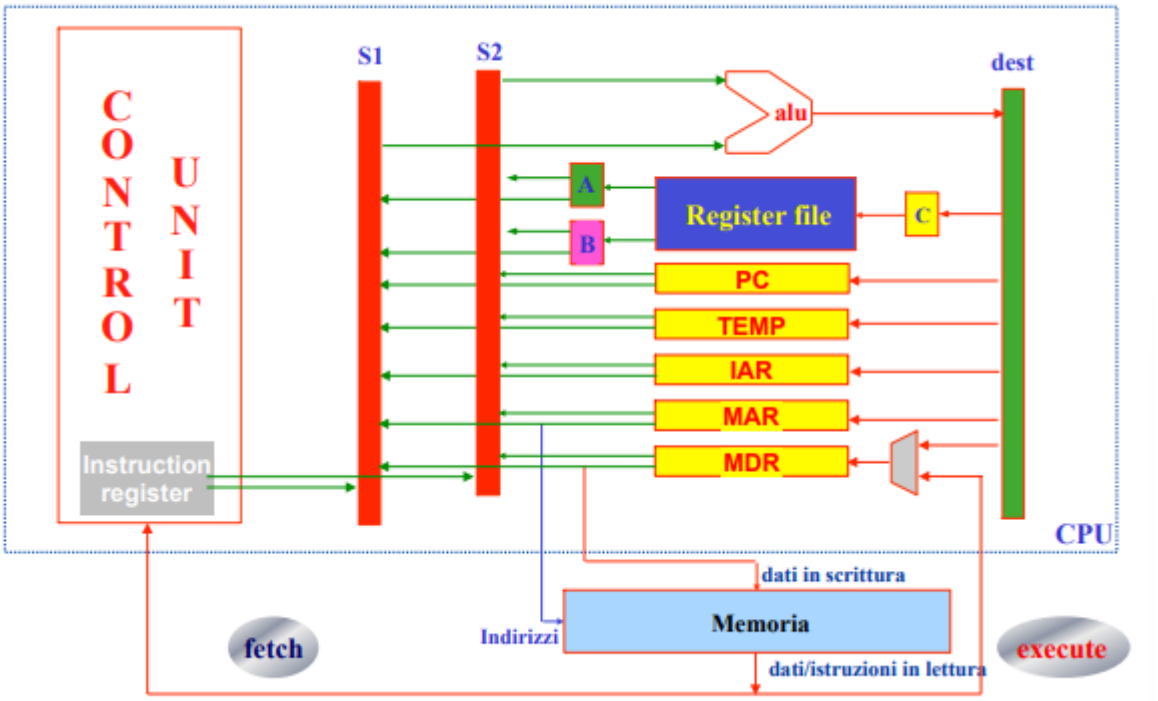
La struttura di una CPU, come tutte le reti logiche sincrone che elaborano dati, può essere strutturata in due blocchi: Unità di controllo e Datapath. Essendo due reti sequenziali hanno bisogno di un segnale di clock.

La CPU, per funzionare, ha bisogno della memoria esterna su cui risiedono il programma e i dati.

DATAPATH – Contiene tutte le unità di elaborazione ed i registri necessari per l'esecuzione delle istruzioni della CPU. Ogni istruzione appartenente all'ISA è eseguita (viene scomposta) mediante una successione di operazioni elementari, dette micro-operazioni. Nel Datapath c'è una ALU che permette di eseguire operazioni aritmetico-logiche.

MICRO-OPERAZIONE – Operazione eseguita all'interno del DATAPATH in un ciclo di clock (es. Trasferimento di un dato da un registro a un altro, elaborazione ALU…).

UNITA' DI CONTROLLO – E' una RSS che in ogni ciclo di clock invia un ben preciso insieme di segnali di controllo al DATAPATH al fine di specificare l'esecuzione di una determinata micro-operazione.



Nel Register File ci sono i 32 registri, che possono essere letti contemporaneamente. Il register file ha due porte di lettura e una di scrittura. Nello stesso clock possiamo leggere da due registri e scrivere in un terzo registro.

L'informazione alla ALU che dice che in quell'istante del clock deve fare quella determinata operazione viene dalla decodifica dell'istruzione. I segnali di controllo arrivano dall'unità di controllo, che ha fatto il fetch dell'istruzione dalla memoria. L'istruzione viene portata nell'instruction register da cui vengono inviati i comandi al datapath per eseguire quella particolare istruzione.

PC – Program Counter – E' il registro nel quale c'è l'indirizzo dell'istruzione che deve essere letta dalla memoria, cioè l'indirizzo nel quale fare il fetch.

MAR – Memory Address Register – Contiene l'indirizzo del dato da scrivere o leggere in memoria e sarà connesso materialmente al bus degli indirizzi.

TEMP – Temporary Register – Registro di deposito temporaneo di risultati

IAR – Interrupt Address Register – E' il registro nel quale viene memorizzato il PC dell'indirizzo nel quale tornare quando si è concluso l'handler di quell'interrupt.

Se volessi prendere ciò che c'è in IAR per farlo finire in PC, dovrei seguire il sentiero del Datapath. Perciò, devo togliere dal tri-state una delle uscite dell'IAR per farla arrivare sul bus (S2) e successivamente farla arrivare all'interno del Program Counter attraverso la ALU. Perciò la ALU deve far passare inalterato l'operando che arriva da S2. Per far ciò, l'unità di controllo deve segnalare alla ALU che in quel momento del clock la ALU deve funzionare in un certo modo.

I dati attraversano la ALU, giungono sul bus di destinazione e arrivano all'ingresso di tutti i registri, compreso il PC. A questo punto, se voglio far arrivare IAR nel PC, l'unità di controllo deve attivare il WRITE ENABLE del registro PC.

MDR – Memory Data Register – Registro di transito temporaneo dei dati da e per la memoria. Ciò che viene letto nel MAR viene scritto nell'MDR

Dei 32 bit presenti nel MAR, solo 30 sono emessi sul bus degli indirizzi, perché BA1 e BA0 non vengono proprio generati per permettere di emettere i segnali BE3 BE2 BE1 BE0.

A e B – Registri di uscita dal Register File

FUNZIONI DELLA ALU

DEST (USCITE) – 4 BIT DI COMANDO

- S1+S2
- S1-S2
- S1 AND S2
- S1 OR S2
- S1 XOR S2
- SHIFT S1 A SINISTRA DI S2 POSIZIONI
- SHIFT S1 A DESTRA DI S2 POSIZIONI
- SHIFT S1 ARITMETICO A DESTRA DI S2 POSIZIONI
- S1
- S2
- 0
- 1

FLAG DI USCITA

- ZERO
- SEGNO NEGATIVO
- CARRY

La ALU è una rete puramente combinatoria. Non esiste nel DLX un registro di FLAG. E' l'unità di controllo che accede ai flag di uscita.

LOAD O STORE

L'indirizzo è sempre formato da registro + immediato a 16 bit esteso a 32. Nel MAR ci andrebbe, nel caso di lettura o scrittura, la somma del registro e dell'immediato che è nell'istruzione. La somma viene eseguita dalla ALU.

Il registro viene estratto dal register file (passando per A/B e per il bus S1/S2). La somma dell'ALU viene scritta nel MAR, che conterrà l'indirizzo definitivo.

TRASFERIMENTO DATI SUL DATAPATH

I bus S1 ed S2 sono multiplexati (tri-state) con parallelismo 32 bit.

I registri campionano sul fronte positivo del clock, hanno due porte di uscita O1 e O2 per i due bus (o i registri A e B) e dispongono di tre ingressi di controllo: un ingresso di Write Enable (WE*) ed uno di Output Enable per ogni porta di uscita, una per ogni bus S1 e S2 (OE1* e OE2*).

Al fine di valutare la massima frequenza a cui è possibile far funzionare il datapath è importante conoscere le seguenti temporizzazioni:

- T_C (max) – ritardo max tra il fronte positivo del clock e l'istante in cui i segnali di controllo generati dall'unità di controllo sono validi.
- T_{OE}(max) – ritardo max tra l'arrivo del segnale OE e l'istante in cui i dati del registro sono disponibili sul bus
- T_{ALU}(max) – ritardo massimo introdotto dalla ALU
- T_{SU} (min) – tempo di setup minimo dei registri (requisito minimo per il corretto campionamento da parte dei registri)

La massima frequenza di funzionamento del datapath si calcola come

$$T_{CLOCK} > T_C(MAX) + T_{OE}(MAX) + T_{ALU}(MAX) + T_{SU}(MAX)$$

$$f_{CK}(max) = \frac{1}{T_{CLOCK}}$$

IL PROGETTO DELL'UNITA' DI CONTROLLO

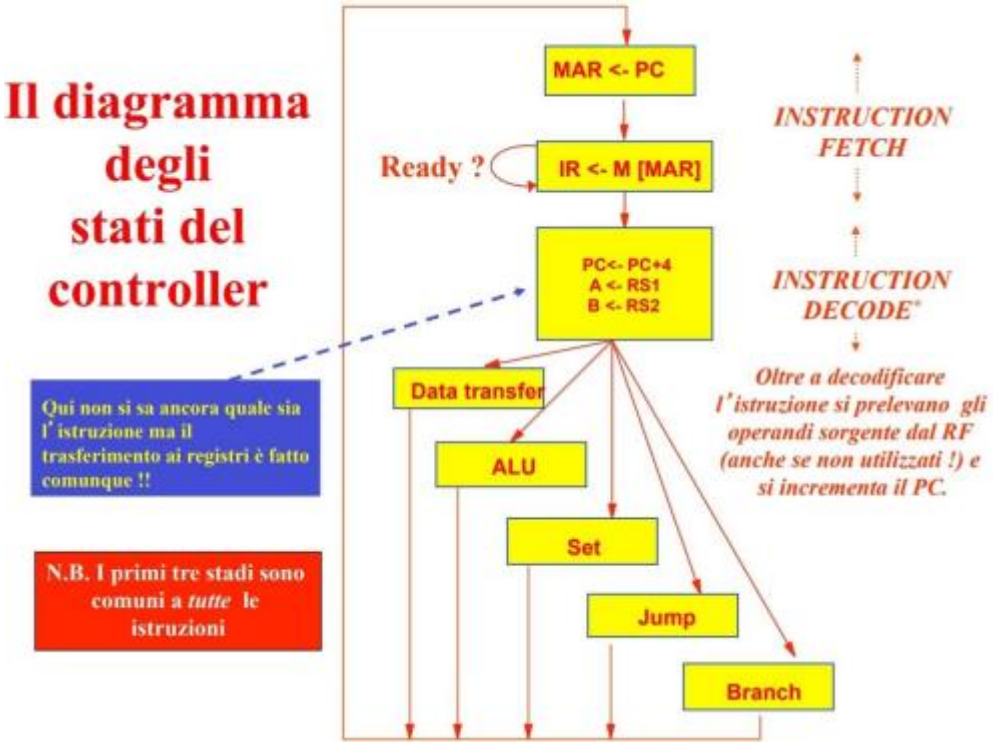
Una volta definito il Set di Istruzioni e progettato il DATAPATH, il passo successivo del progetto di una CPU è il progetto dell'unità di controllo.

Il CONTROLLER è una RSS: il suo funzionamento può essere specificato tramite un diagramma degli stati. Il controller permane in un determinato stato per un ciclo di clock e transita da uno stato all'altro in corrispondenza degli istanti di sincronismo (fronti del clock).

Ad ogni stato corrisponde quindi un ciclo di clock. Le micro-operazioni che devono essere eseguite in quel ciclo di clock sono specificate nel diagramma degli stati che descrive il funzionamento del CONTROLLER all'interno degli stati.

A partire dalla descrizione RTL si sintetizzano poi i segnali di controllo che devono essere inviati al DATAPATH per eseguire le operazioni elementari associate a ogni stato.

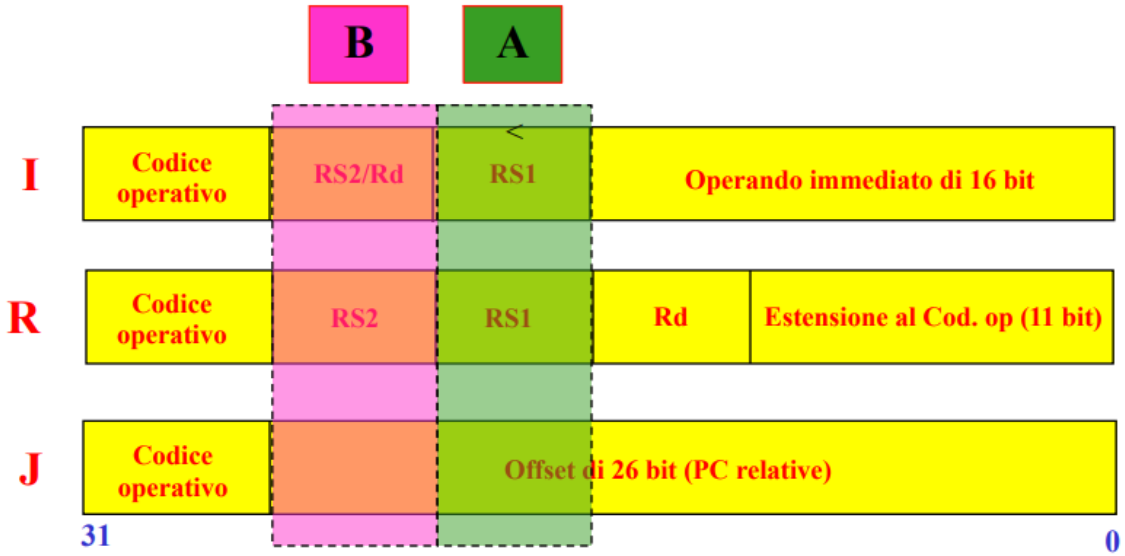
DIAGRAMMA DEGLI STATI DEL CONTROLLER



Quando l'istruzione arriva a IR (Instruction Register interno all'unità di controllo), deve essere decodificata.

Per anticipare le operazioni l'unità di controllo in maniera arbitraria estrae due dati dal register file, che sono potenzialmente utili allo svolgimento dell'operazione. I dati estratti vengono temporaneamente salvati in A e B, registri di supporto.

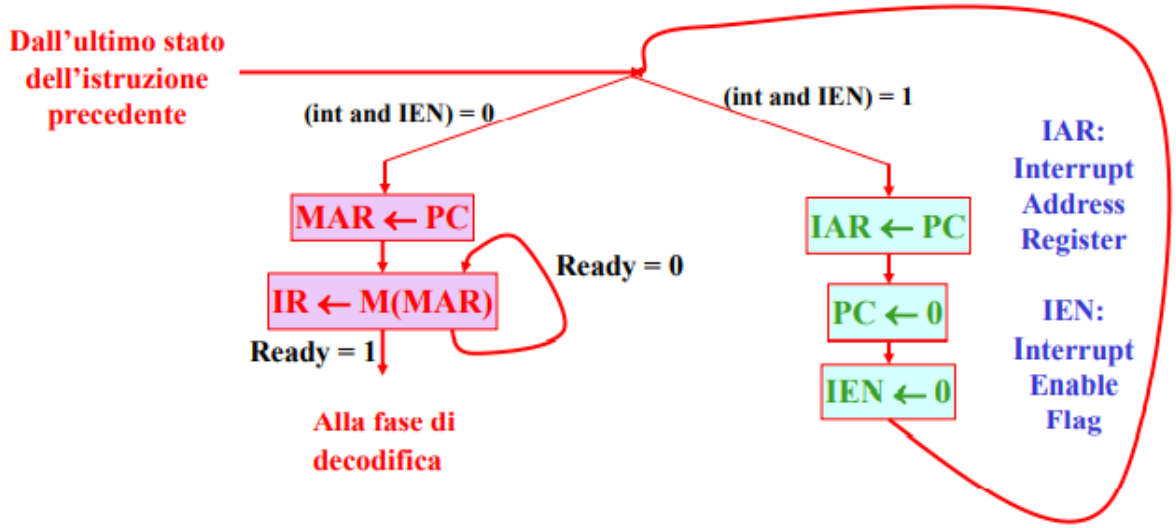
ESTRAZIONE AUTOMATICA DEI REGISTRI DURANTE LA FASE DI DECODE DI UN'ISTRUZIONE



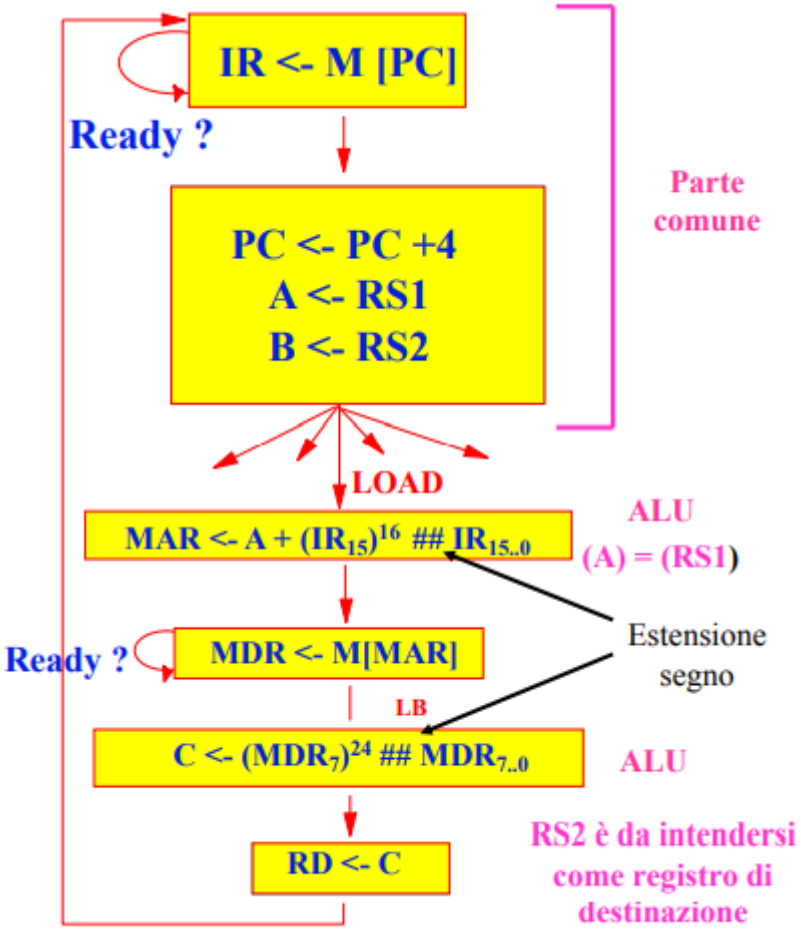
Questi 5+5 bit sono utilizzati per estrarre, preventivamente e ancora prima di conoscere che tipo di istruzione è stata letta dalla memoria, dal register file due registri in A e B. Nel caso di un'istruzione J non ci sono registri coinvolti e quindi saranno estratti bit corrispondenti all'offset. Nel caso di istruzione I, in B potrebbe finire il valore del registro destinazione. Infine, i 5+5 bit rappresentano gli indici, ma non il valore dei due registri che è contenuto nel register file.

GLI STATI DELLA FASE DI FETCH

In questa fase si deve verificare se è presente un interrupt. Se l'interrupt è presente e può essere servito (IEN = true) si esegue implicitamente l'istruzione di chiamata a procedura all'indirizzo 0, e si salva l'indirizzo di ritorno nell'apposito registro IAR. Se l'interrupt non è presente o le interruzioni non sono abilitate, si va a leggere in memoria la prossima istruzione da eseguire (il cui indirizzo è in PC).



CONTROLLO PER L'ISTRUZIONE LOAD BYTE



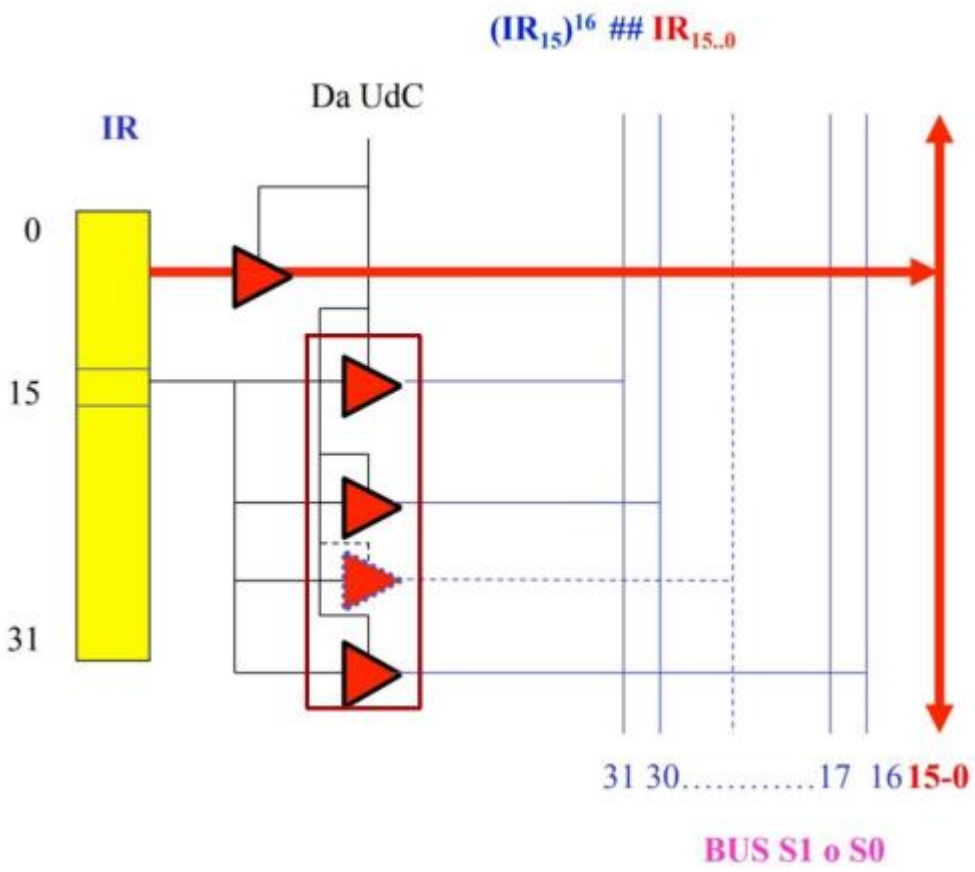
Durante il primo clock viene letta dalla memoria l'istruzione all'indirizzo nel pc. Durante il secondo clock, in contemporanea alla decodifica dell'istruzione dell'unità di controllo, in parallelo si aggiorna il program counter perché l'istruzione successiva a meno di un interrupt si troverà in PC+4.

L'unità di controllo del dlx, al termine di questa fase, sa quale istruzione deve eseguire. Se è una load, vengono eseguite tutte le istruzioni per realizzare tale scopo.

Viene costruito l'indirizzo: il registro sorgente viene sommato all'immediato presente nell'istruzione, che viene esteso con segno attraverso una rete opportuna, attraverso la ALU. Il risultato viene memorizzato in MAR al fronte del clock.

Il contenuto della memoria all'indirizzo presente nel valore nel MAR viene letto e il contenuto finisce nell'MDR. Quando READY = 1, l'operazione è terminata, altrimenti c'è un nuovo ciclo di clock.

ESTENSIONE DEL SEGNO



Per l'estensione del segno ho bisogno di tutti i tri-state, perché usando solo l'i-esimo tri-state, in altre situazioni manderei in cortocircuito il bus.

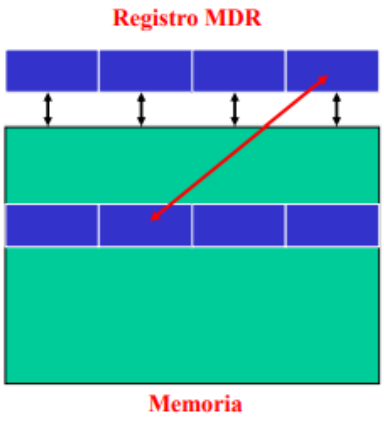
CONTROLLO PER LE ISTRUZIONI DI DATA TRANSFER

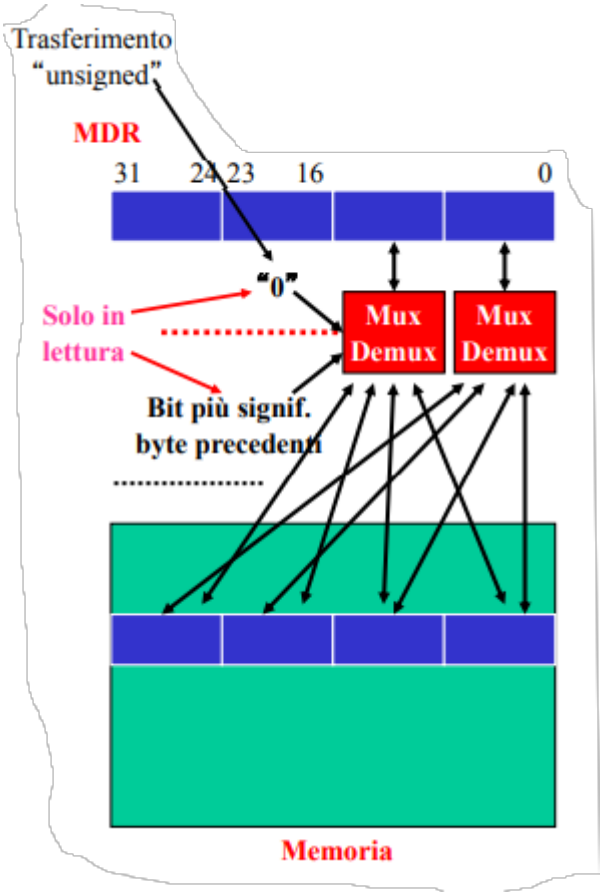
1) Estensione immediato con segno e somma con registro che viene messo in MAR
2) In caso di LOAD: lettura e poi estensione di MDR.
3) In caso di scrittura: copia del valore in MDR e poi scrittura del valore in memoria

TRASFERIMENTI BYTE E HALFWORD E WORD

I trasferimenti di bytes sono sempre considerati allineati. I trasferimenti di half-word devono avvenire a indirizzi multipli di 2, mentre quelli di word a indirizzi multipli di 4. In caso di disallineamento, il trasferimento fallisce. La lettura/scrittura di bytes e half-word (a causa del reciproco disallineamento fra i registri e la memoria) implica che fra i registri e la memoria siano interposti dei mux/demux (realizzati con tri-state).

Nel caso di STORE non si ha mai estensione del segno, che avviene solo per letture di 8/16 bit

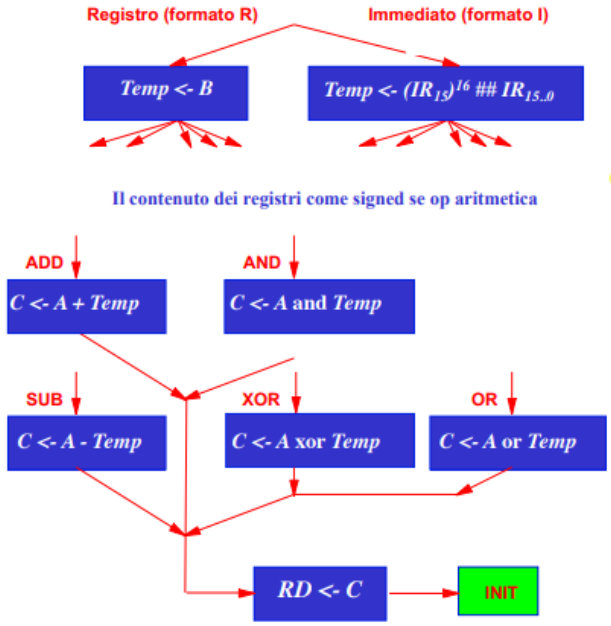




I mux 23-16 e 31-24 hanno come ingresso anche il bit 7 del byte 7-0 della memoria (LB) e il bit 15 del byte 15-8 della memoria (LH).

Ad esempio, in una LB il MUX 7-0 si collega direttamente alla memoria mentre i MUX 15-8, 23-16 e 31-24 si collegano al bit 7 del MUX 7-0 proveniente dalla memoria. In una SH a indirizzo multiplo di 2 e non di 4 il DEMUX 7-0 dal MDR si collega alla memoria 23-16 e il DEMUX 15-8 alla memoria 31-24. Gli altri due byte di memoria rimangono invariati.

ISTRUZIONI ALU

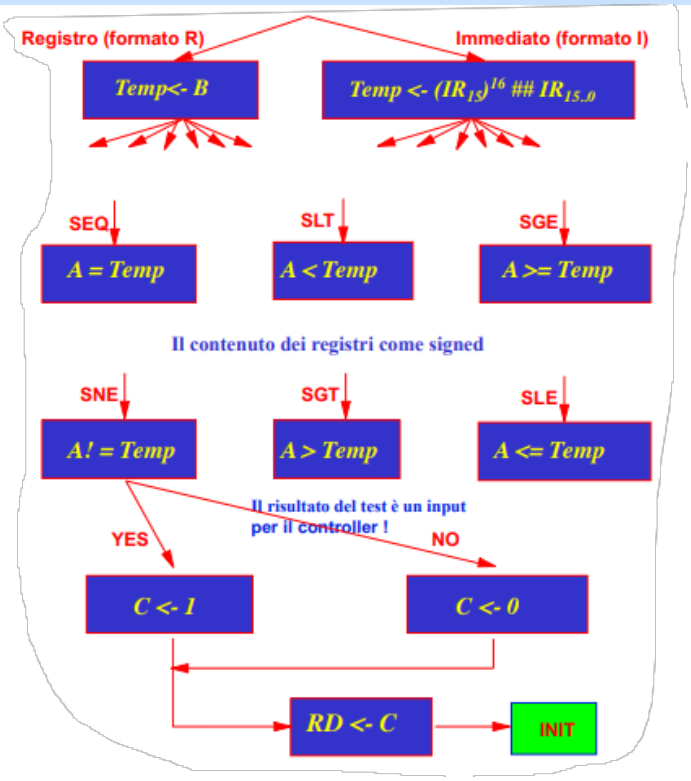


Istruzioni tra 2 registri, oppure un registro e un immediato. Per uniformare il diagramma degli stati, si usa TEMP (registro temporaneo). Se l'istruzione è di tipo I, viene messo il valore dell'immediato a 16 bit esteso con segno a 32 bit. Se l'istruzione è in formato normale viene messo il valore contenuto nel registro TEMP.

E' un registro ausiliario finalizzato a ciò, non necessario ma utile. Dopo l'uniformazione, a volte viene peggiorata l'efficienza. Ciò vale per ogni operazione aritmetica.

Il MAR non è coinvolto perché non c'è accesso alla memoria, idem per MDR. Ciò consente di uniformare i diagrammi degli stati.

ISTRUZIONI DI SET (CONFRONTO)



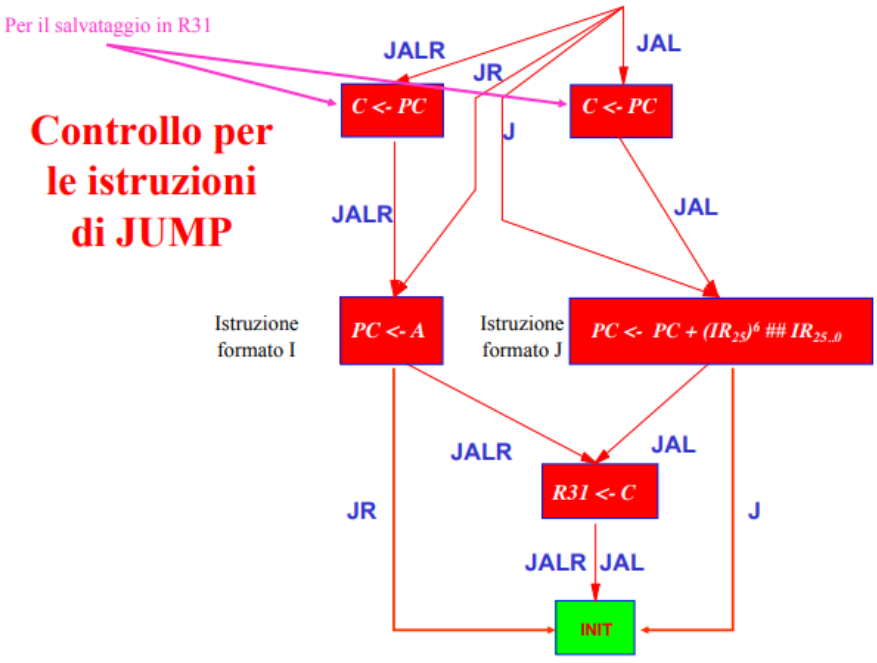
Se dovessi fare un confronto tra registri sorgente non sarebbe necessario TEMP. L'unità di controllo potrebbe programmare la ALU per fare il confronto.

Se l'istruzione prevede un immediato, bisogna passare da TEMP. In TEMP viene messo l'immediato esteso a 32 bit.

Da lì in poi si può fare il diagramma considerando 2 registri e non un registro e un immediato.

Per i salti non condizionati si usa JUMP

ISTRUZIONI DI JUMP



JALR / JR – Jump con valore salvato in un registro

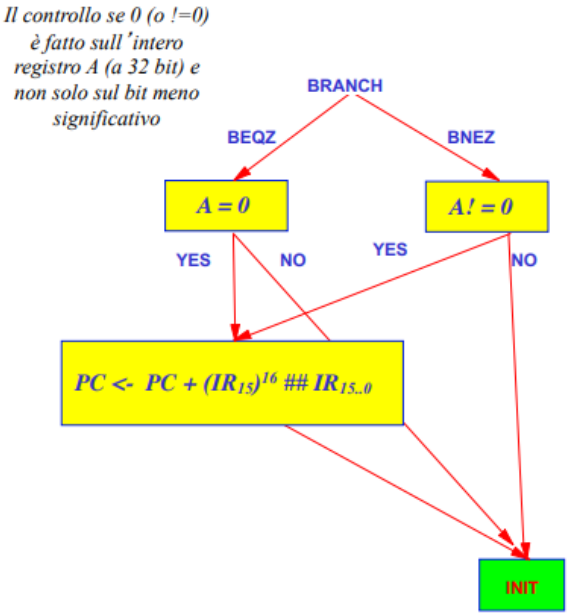
JAL / JMP – Jump con valore in un immediato (Rappresenta l'offset di quanto saltare).

Se si effettua una Jump And Link si salva in R31 il contenuto di C (ossia il PC da cui si era partiti). A meno di interrupt, il prossimo fetch sarà fatto a R31.

ISTRUZIONI DI BRANCH

Controllo per le istruzioni di BRANCH

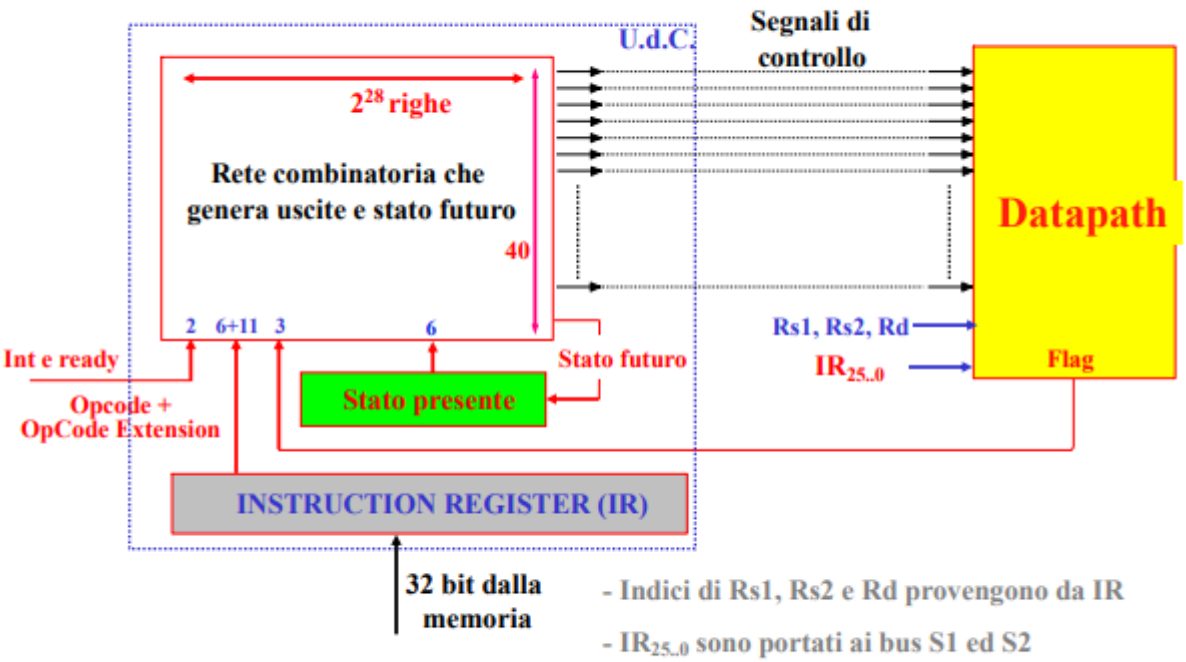
Ex. BNEQZ R5, 100



Viene effettuato un salto se il registro è o non è zero. Per calcolare l'indirizzo il PC+4 viene sommato con l'immediato a 16 bit (esteso a 32 bit) dalla ALU.

Dopodiché viene aggiornato il valore contenuto nel PC. Se la condizione non è verificata il valore del PC non si aggiorna.

Controllo cablato ("*hardwired*")



PASSI DELL'ESECUZIONE DELLE ISTRUZIONI

Nel DLX l'esecuzione di tutte le istruzioni può essere scomposta in 5 passi, ciascuno eseguito in uno o più cicli di clock.

1) FETCH – L'istruzione viene prelevata dalla memoria e posta in IR
2) DECODE – L'istruzione IR viene decodificata e vengono prelevati gli operandi sorgente del Register File
3) EXECUTE – Elaborazione aritmetica o logica mediante la ALU
4) MEMORY – Accesso alla memoria e, nel caso di BRANCH, aggiornamento del PC.
5) WRITE-BACK – Scrittura sul register file

Queste 5 fasi sono gli stadi della nostra catena di montaggio. Organizzeremo il DATAPATH e il WORKFLOW affinché possano avvenire in modo sequenziale e indipendente. Ogni fase è sempre attiva e opera su istruzioni diverse.

1) FETCH

MAR ← PC ;

IR ← M[MAR];

2) DECODE

A← RS1, B← RS2, PC← PC+4

3) EXECUTE

- **MEMORIA:**

MAR ← A + $(IR_{15})^{16}$ ## $IR_{15..0}$; **(utilizzano ALU, S1, S2, dest)**

MDR← B; ***(NB: serve nelle Store ove RD=RS2 operazione non significativa nelle LOAD)***

- **ALU:**

C <- A *op* B (*oppure* A op $(IR_{15})^{16}$ ## $IR_{15..0}$) ;

C <- sign(A *op* B (*oppure* A op $(IR_{15})^{16}$ ## $IR_{15..0}$)); se SCn

- **BRANCH:**

Temp ← PC + $(IR_{15})^{16}$ ## $IR_{15..0}$) ; **(utilizza ALU, S1, S2, dest: qui non si sa ancora se si deve saltare)**

- **J e JAL**

Temp ← PC + $(IR_{25})^{6}$ ## $IR_{25..0}$) ;

- **JR e JALR**

Temp ← A;

4) MEMORY

- **Memoria:**

MDR ← M[MAR]; (*LOAD*)

M[MAR] ← MDR; (*STORE*)

- **BRANCH:**

If (Cond) PC ← Temp;

[A] è il registro che condiziona il salto (Cond) ;

- **JAL e JALR:**

C ← PC;

5) WRITE-BACK

- **istruzioni diverse da J, JR, JAL, JALR**

C ← MDR; (*se è una LOAD – due micropassi*))

RD ← C ;

- **istruzioni J, JR, JAL, JALR**

PC ← Temp;

RD ← C ;